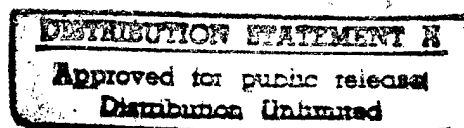


Proceedings

IFIP WG 11.3

Eleventh Annual Working Conference



on

Database Security



DTIC QUALITY INSPECTED 2

San José State
UNIVERSITY



11-13 August 1997
Lake Tahoe, California

19980203 025

Proceedings

IFIP WG 11.3

Eleventh Annual Working Conference

on

Database Security

**11-13 August 1997
Lake Tahoe, California**

ACKNOWLEDGMENT

We think the following people for making the Eleventh Annual IFIP WG 11.3 Working Conference in Database Security a great Success:

The authors of the papers submitted to the conference,

The reviewers of the papers,

David Spooner, the chair of the IFIP 11.3. Working Group, for his encouragement throughout the organization of the conference and for his support in putting the final program together,

Teresa Lunt, for his activities as the general chair of the conference,

The participants of the conference,

The sponsors of the conferences

T. Y. Lin and Shelly Qian
Program co-Chairs

Eleventh IFIP WG 11.3 Working Conference Committee

Program co-Chairs

T. Y. Lin
Department of Mathematics and
Computer Science
San Jose State University
129 S. 10th St,
San Jose, California 95192 USA
Tel: +1-408-924-5121
Fax: +1-408-924-5080
E-mail: tylin@cs.sjsu.edu

Xiaolei Qian
SecureSoft, Inc.
275 Shoreline Dr., Suite 520
Redwood Shores, CA 94065 USA
Tel: +1-415-596-2425
Fax: +1-415-596-0503
E-mail: sqian@acm.org

General Chair

Teresa Lunt
DARPA/ITO
3701 N Fairfax Dr
Arlington, Virginia 22203 USA
Tel: +1-703-696-4469
Fax: +1-703-696-2202
E-mail: tlunt@darpa.mil

IFIP WG11.3 Chair

David L. Spooner
Computer Science Department
107 Amos Eaton
Rensselaer Polytechnic Institute
Troy, New York 12180-3590 USA
Tel +1-518-276-6890
Fax: +1-518-276-4033
E-mail: spoonerd@cs.rpi.edu

Table of Contents

Panel I

1. Data Warehousing, Data Mining, and Security
Moderator: Bhavani Thursingham

Policy Modeling

Chair: Sushil Jajodia

2. Access Control by Object-Oriented Concepts1
Wolfgang Essmayr , G. Pernul , A M. Tjoa
3. Administration Policies in a Multipolicy Authorization System.....15
Elisa Bertino, Elena Ferrari

Invited Talk

Chair: Teresa Lunt

4. Security Issues in Integrated Storage
Jose A. Blakeley

Distributed and Federated Systems

Chair: T. C. Ting

5. Designing Security Agents for the DOK Federated System31
Zahir Tari
6. Web Implementation of a Security Mediator for Medical Databases.....47
G. Wiederhold, M. Bilello, C. Donahue
7. Supporting the Requirements for Multilevel Secure and Real-time
Databases in Distributed Environments.....57
Craig Chaney, Sang H. Son

Objected-Oriented Systems

Chair: Ravi S. Sandhu

8. A Principled Approach to Object Deletion and Garbage Collection in a Multi-level Secure Object Base.....75
Elisa Bertino, Elan Ferrari
9. Compile-time Flow Analysis of Transactions and Methods in Object-Oriented Databases.....88
Masha Gendler-Fishman, Ehud Gudes
10. Capability-based Primitives for Access Control in Object-Oriented systems.....104
John Hale, Jody Threet, Sujeet Sheno

Panel II

11. CORBA Security
Moderator: Catherine McCollum

Work Flow

Chair: John McDermott

12. An Execution Model for Multilevel Secure Workflows.....120
Vijayalakshmi Atluri, Wei-Kuang Huang Elisa Bertino
13. Task-based Authorization Controls (TBAC): Models for Active and Enterprise-oriented Authorization Management.....136
Roshan K. Thomas, Ravi S. Sandhu
14. Alter-egos and Roles Supporting Workflow Security in Cyberspace.....152
Ehud Gudes, Reind P. van de Riet, Hans FM Burg, Martin Olivier

Architectures and Systems

Chair: Sujeet Sheno

15. A Two-tier Coarse Indexing Scheme for MLS Database Systems.....170
Sushil Jajodia, Ravi Mukkamala, Indrajit Ray
16. Replication Does Survive Information Warfare Attacks.....186
John McDermott

17. Priority-driven Secure Multiversion Locking Protocol for
Real-Time Secure Database Systems.....200
Chanjung Park, Seog Park , Sang H. Son

Multilevel Security

Chair: Pierangela Samarati

18. IRI: A Quantitative Approach to Inference Analysis in
Relational Databases.....214
Kan Zhang
19. Hierarchical Namespaces in Secure Databases
Adrian Spalka, Armin B. Cremers (University of Bonn)

Role-based Access Control

Chair: Elisa Bertino

20. Software Architectures for Consistency and Assurance of
User-Role Based Security Policies.....223
S. Demurjian, T. C. Ting, J. Reisner
21. Role-Based Administration of User-Role Assignment:
The URA97 Model and its Oracle Implementation.....239
Ravi Sandhu, Venkata Bhamidipati (George Mason University)

Short Talks

22. Advanced Internet Search Tools: Trick or Treat?257
G. Lorenz, S.Dangi, D. Jones, P. Carpenter, G. Manes, S. Sheno
23. An Enviroment for Developiiing Securely Interoperable Heterogeneous
Distributed Objects258
M. Berryman, C. Rummel, M. Papa, J. Hale, J. Threet, S. Sheno
24. Multilevel Decision Logic: A Formalism for Multilevel Rules Mining259
Xiaoling Zuo, T. Y. Lin

Panel I

Data Warehousing, Data Mining, and Security
Moderator: Bhavani Thursingham

Policy Modeling
Chair: Sushil Jajodia

Access Controls by Object-Oriented Concepts

W. Essmayr¹⁾, G. Pernul²⁾, A. M. Tjoa³⁾

¹⁾ Research Institute for Applied Knowledge Processing
Softwarepark Hagenberg
Hauptstraße 99, A-4232 Hagenberg, Austria
e-mail: we@faw.uni-linz.ac.at

²⁾ Department of Information Systems
University of Essen
Altendorfer Straße 97, D-45143 Essen, Germany
e-mail: pernul@wi-inf.uni-essen.de

³⁾ Institute of Software Technology
Technical University of Vienna
Resselgasse 3, A-1040 Vienna, Austria
e-mail: tjoa@ifs.tuwien.ac.at

Abstract

This paper introduces *object-oriented access controls* (OOAC) as a result of consequently applying the object-oriented paradigm for providing access controls in object and interoperable databases. OOAC includes: (1) subjects, like users, roles etc., are regarded as first-class objects, (2) objects are accessed by sending messages, and (3) access controls deal with controlling the flow of messages among objects. OOAC are not intended to replace legacy access control mechanisms which mainly have been designed and applied in non-object environments. Instead, they provide the basis for applying these concepts in true object-oriented environments. An *object authorization language* (OAL) is proposed for specifying authorizations in a declarative manner. We illustrate the feasibility of the proposed concepts in applying them to IRO-DB II, an extension of the database federation IRO-DB, that provides interoperable access between relational and object-oriented database systems on the world-wide-web.

Keywords

security policy, object-orientation, access controls, interoperability

1 Introduction

The importance of object-oriented database systems increased dramatically within the commercial database market in the last few years. Especially, new application domains, like multimedia or interoperable environments, illustrate the feasibility and usefulness of object-orientation. Concerning access controls, most of the existing models, for instance *discretionary access controls* (DAC), *role-based access controls* (RBAC), or *mandatory access controls* (MAC), have been originally designed for relational database systems. However, the application of these models in object-oriented systems can not be straight forward (compare [6]). Particular object-oriented characteristics, like object identity, encapsulation, inheritance, polymorphism, and complex objects (see [1]) require the integration of new mechanisms to legacy access control concepts.

A remarkable amount of research has been devoted to extending access control concepts for object environments (see section 1.1). All this work offers the possibility to understand the challenges and research issues concerning access controls within object-oriented systems. However, we see the need for a common ground to start from in order to develop *true* object-oriented access controls. This is mainly because of two reasons: (1) to the best of our knowledge, none of the proposed extensions to legacy access controls consider subjects (e.g. users) to be first-class objects within an object database. Instead, subjects are treated as "extra-terrestrial" entities that are completely separated from the data objects. In consequence, object-oriented features can not be applied to subjects, respectively, access control concepts can not be applied within pure data object communications. On the other hand, (2) most of the proposed extensions still regard access types as a set of elementary actions (like *read*, *write*, *execute*, etc.). This attitude ignores *messages* to be the means of communication within an object system and dramatically limits the expressiveness of the security model. The set of messages an object is able to respond to (i.e. the object's interface) exactly defines the ways an object might be accessed; a set of elementary actions can never be complete for the great variety of application domains.

The remainder of the paper is structured as follows: section 2 summarizes basic terminology and provides an overview about the database federation IRO-DB II. Section 3 introduces the concepts used by object-oriented access controls. It concentrates on the object activity stack, proposes an object authorization language, and specifies policies concerning authorization and access control within OOAC. Finally, section 4 concludes and provides directions for future research efforts.

1.1 Related Work

In [3], the authors present an authorization model for next-generation database systems supporting object-oriented concepts as well as semantic data modeling concepts. Special effort is given to the development of computing implicit authorizations from a set of explicitly defined authorizations. [5] first suggests to

specify privileges for users to execute methods on objects. The authorization model enforces the concept of private and protected methods. [14] presents a method-based authorization model as well as algorithms that evaluate the proposed authorization policies concerning generalization, aggregation, relationships, abstract classes and indirect method-calls. [6] summarizes the issues arising within discretionary authorizations for object bases. The authors provide suggestions how to successively extend a basic authorization model with object-oriented features concerning access types, security subjects and objects, access controls, and authorization. [16] concentrates on the evaluation algorithms deciding whether an access should be fully or partially granted/denied. The algorithms are discussed for compiled-time as well as run-time evaluation. [15] develops an authorization model for object-oriented databases. The model contains user access controls and administration of authorizations. It consists of a set of policies, a structure for authorization rules and algorithms to evaluate access requests against the authorization rules. [17] discusses means for object-oriented environments to reduce the number of explicit authorizations to the number of meaningful authorizations. In this work the concept of compound subjects is introduced in order to be able to model certain real world security requirements (i.e. the four eyes principle). [7] provides another early but important contribution to authorizations in object-oriented environments. [11] discusses role-based access controls in an interoperable environment, including object-oriented as well as relational database systems.

2 Basic Terminology, IRO-DB Overview

The basic characteristics that are common to every object-oriented system can be summarized as follows (compare [1], or [4]):

- *object*: a real-life entity having *structure*, *state* and *behavior*. Each object is associated with an *object identifier* (oid) that is unique and fixed for the whole life of the object.
- *structure* and *state*: each object is associated with a set of *attributes* that specify the structure of an object. The *attribute values* being themselves

objects determine the state of an object at any time.

- *behavior*: each object is associated with a set of *methods* specifying an object's behavior. A method consists of a method name, a *signature* (the names and types of parameters and the result type) and an *implementation* (a piece of executable code) which can be invoked by a *message* that matches a particular method name and signature. Some systems allow a method to be *overloaded* meaning that another method of that object has the same name but different signature.
- *encapsulation*: the attributes of an object can only be accessed by sending messages to the object resulting in the execution of a method corresponding to the particular message. The set of messages an object can respond to is called the object's *interface*. The execution of a method may imply the object sending messages to itself or to some other objects.
- *class*: a template for a set of objects sharing the same *structure* and *behavior*. Additionally, classes serve as an *object factory* (create objects of a class by sending the *new* message) and an *object warehouse* (maintaining the extent, i.e. the set of objects that are instances of a class).
- *inheritance*: classes can have more specialized sub-classes and more general super-classes with the purpose that all sub-classes inherit the super-class' structure and behavior. Sub-classes may define additional attributes and methods or may override inherited methods with the impact that objects of different levels of an inheritance hierarchy have methods with the same name and signature but different behavior.
- *complex objects*: objects that are built from simpler ones by applying complex object constructors to them including *sets*, *lists*, *bags*, *arrays*, *tuples*, and the like.

Any object-oriented database system has to provide the features and characteristics listed above combined with *extensibility* (no distinction in usage between system defined and user defined types), *computational completeness* (ability to express any computational function), and typical database features like *persistence*, *secondary storage management*, *recovery mechanisms*, and *ad hoc query facilities*. In

this paper we do not concentrate on the optional and open features of an object-oriented database system as mentioned in [1] since these characteristics (e.g. multiple inheritance, versions, etc.) still differ significantly among current object databases.

2.1 The IRO-DB II Database Federation

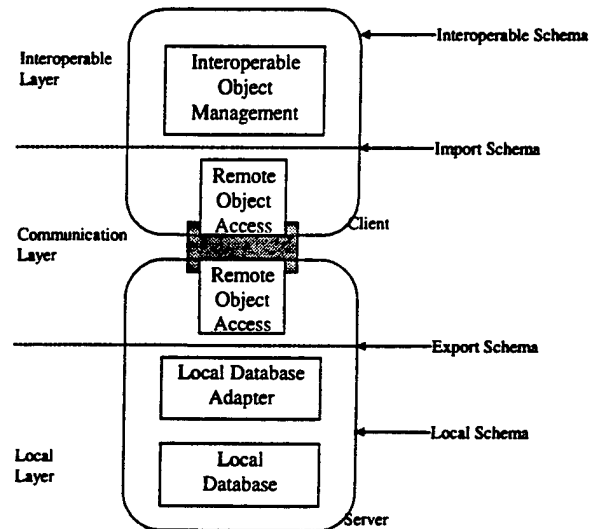


Figure 1: The IRO-DB architecture.

IRO-DB (interoperable relational and object-oriented databases) is a European ESPRIT project implementing a database federation (compare [20]). It uses a three layered architecture (see Figure 1) having a *local*, a *communication*, and an *interoperable* layer. A common object-oriented data model (ODMG, see [2]) is used throughout the layers in order to provide interoperability. The model defines an *object definition language* to declare the interfaces to object types, an *object query language* to formulate database queries, and an *object manipulation language* to retrieve and manipulate database objects within a programming language (e.g. C++). All component databases at the local layer of IRO-DB implement a so called *local database adapter* (LDA) which makes the local system appear as if it was an ODMG compliant database. As shown in Figure 1, each LDA exports its *local schema* or parts of it as an *export schema* using *remote object access* services of the communication layer. The various export schemata are imported at the interoperable layer (*import schema*) and integrated into an *interoperable schema* using *derived classes* (compare [8]). These kind of classes provide federated views on the classes of the

import schema with a unique semantic for all object types and methods.

The authorization model specified within IRO-DB provides a high level of security. It combines ownership of data with centralized administration of security issues by using role-based access controls. Furthermore, powerful concepts like *negative* and *implied* authorization (see [12]) are used to provide enough flexibility for integrating heterogeneous authorization models. The security architecture of IRO-DB (see [13]) allows to specify a global security policy in order to prevent global security risks like multi-site aggregation of data and allows to integrate any number of heterogeneous local security policies for ensuring local autonomy.

IRO-DB II will be an extension of IRO-DB applying the developed database federation within the world-wide-web. An ODMG/JAVA binding is currently under development which will replace the ODMG/C++ binding used in IRO-DB.

3 Object-Oriented Access Controls (OOAC)

In this section we will introduce the concepts developed for OOAC as the basic access control policy of IRO-DB II.

3.1 Prerequisites for OOAC

OOAC requires an object database to provide a class hierarchy having a distinct origin class and a sub-hierarchy of objects that could be referenced by name. Since not every object should be an issue for access controls OOAC assume that only access to *named* objects is being controlled. Some of the object databases explicitly distinguish between *persistent* objects (objects that survive the process within they have been created) and *transient* objects (objects that live only for the execution time of the process within they have been created). Some others implement the concept of *persistence by reference* (an object is persistent as long as it is referenced by at least one other object). Anyway, only *persistent* objects should be an issue for access controls since they are the asset to protect while transient objects 'die' after the scope of a process. In this paper we use the term *protection*

object for objects that could be named and should be an issue for access controls.

The set of classes offered by an object database can be grouped into several schemata, for instance, a *basic* schema, a *security* sub-schema, an *application domain* sub-schema etc. The basic schema as required for OOAC could look like shown in

Figure 2. It contains a root class (*Object*) holding the object identifier and offering methods to create (*new*), *delete*, and *copy* objects. Furthermore, a root class for all named objects (*NamedObject*) is offered, providing methods to find (*lookup*) an object and to retrieve or change its *name*. Finally, some general purpose classes could be provided that are implementing common and useful functionality (e.g. collections, strings, etc.).

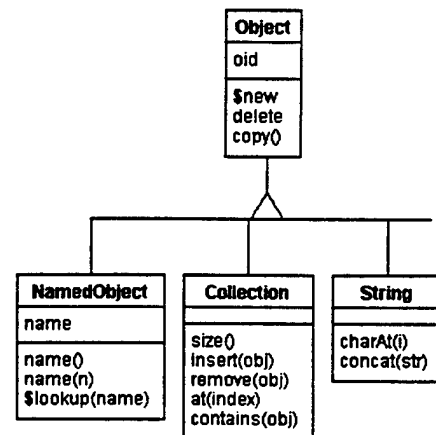


Figure 2: A basic schema as required by OOAC (OMT notation, see [19]).

In a true object-oriented environment basically everything is regarded as an object. Hence, the active entities referred to as *subjects* in security literature (e.g. users, roles, etc.) are objects, too. Figure 3 illustrates a security sub-schema that is apt to implement role-based access controls. Class *Subject* (sub-class of *NamedObject*) provides special behavior concerning activity (see section 3.3) common to all security subjects. Two kinds of subjects exist in the case, namely, *users* and *roles*. Users have to be authenticated (using a password check) and can be members of several roles. A user has to *play* (activate) a role in order to receive certain authorizations. Roles may be structured within a role-hierarchy using the *subroles/superrole* relationships.

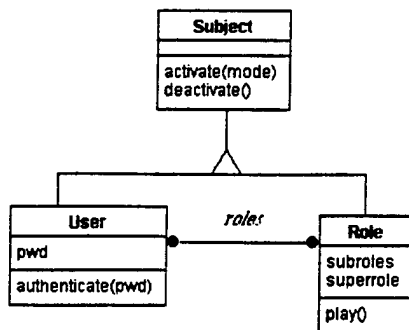


Figure 3: An example for a security sub-schema for role-based access controls.

An application sub-schema contains classes relevant to a particular application domain. In this paper, we use a simple example used within the IRO-DB project. It focuses on a production database that maintains *parts* which are distributed over two local databases.

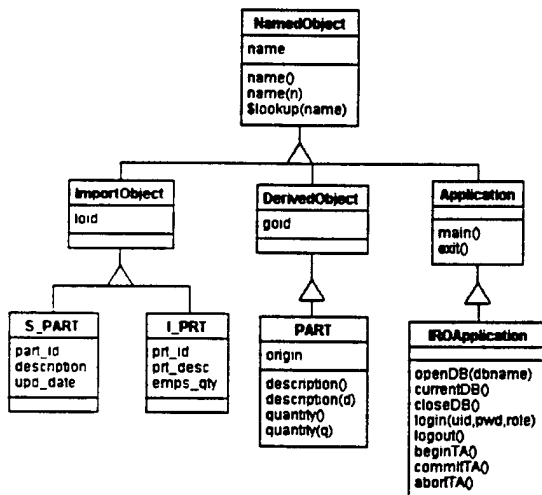


Figure 4: An example for an application sub-schema.

Basically, objects at the interoperable layer of IRO-DB may be either *native* objects defined within the home database of the interoperable layer, may be *imported* from a local database, or may be *derived* from native, imported, or other derived objects. Import objects additionally maintain a local object identifier (*loid*) that is composed of the original object identifier and some local site information. Derived objects additionally maintain a global object identifier (*goid*) that contains information about the classes from which the object is derived. The example schema in Figure 4 shows two import classes, namely

S_PART and *I_PRT* with their objects in fact located at different local databases. Both classes maintain a part identifier (*S_PART.part_id*, *I_PRT.prt_id*) and a part description (*S_PART.description*, *I_PRT.prt_desc*) about logically identical parts. However, class *S_PART* additionally maintains an update date (*upd_date*), and class *I_PRT* additionally holds the quantity (*emps_qty*) of produced parts. The derived class *PART* provides a global view on both import classes allowing to retrieve and change the part *description* as well as the *quantity* of any part stored in the distributed local databases. The native class *IROApplication* provides functionality for opening and closing IRO-DB (*openDB*, *closeDB*), for transaction management (*beginTA*, *commitTA*, *closeTA*) and for identification and authentication (*login*, *logout*) of IRO-DB users.

3.2 Messages

As mentioned in section 2, objects communicate in sending messages to themselves or to other objects which react in executing a particular method that shows behavior in that it may change the state of the object, alter parameter values or provide a return value. Object-oriented access controls simply deal with controlling the flow of messages among objects of an object database.

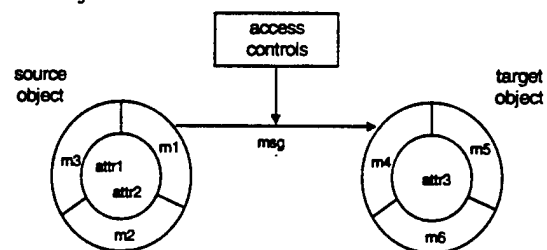


Figure 5: Controlling the flow of messages among objects.

Figure 5 illustrates the message sending and access control mechanisms. During the execution of method *m1* the source object sends message *msg4* to the target object which is expected to react in executing its corresponding method *m4*. OOAC intercepts the message sending process¹ and applies access controls in that it evaluates whether the source object is

¹ Triggers could be an adequate mechanism to intercept the sending of messages among objects.

allowed to send the particular message to the target object (see section 3.5 for details about the evaluation process). If allowed, the message is sent and the target object executes the requested method. If denied, the message is blocked and an *access control exception* is raised². The requested method is prevented from executing and its output parameter values as well as an optional result value are set to a distinct value, e.g. *nil*. Each application may catch the access control exceptions and handle them according to its needs. Several nested method calls may occur during program execution, e.g. *object1* sends *message1* to *object2* which in turn sends *message2* to *object3* etc. which leads to a stack of active objects as explained in the following sub-section.

3.3 Activity

OOAC require a system maintained *activity stack* which usually corresponds to the stack of method calls. Each time, an object receives a message and starts executing the corresponding method, the object becomes *active* and is pushed on to the activity stack. All subsequent messages are regarded to be sent from this object until it returns from executing the method and is popped again from the stack. The most recently activated object is the basis for access control decisions. If a decision is not possible the previously activated object is examined until the primarily activated object is reached which usually corresponds to a kind of default system object. Instances of class *Subject* or sub-classes of it are the only objects that have the possibility to *actively* modify the activity stack independently from the sequence of method invocations in one of the following ways:

- activate *on-behalf-of*: the subject is additionally pushed on to the activity stack which is the normal case like for any other object with the difference that the subject remains activated until it is explicitly deactivated (popped from the stack).
- activate *instead-of*: the subject replaces the previously activated object in the activity stack with the consequence that the authorizations of the

replaced object do not further influence access control decisions.

Table 1 shows the changes to an activity stack during the execution of a simple example application that uses the schema illustrated in Figure 4. The IRO-DB application (IROApplication[1], the characters '[' and ']' denote instantiation, i.e. the object named "1" of class *IROApplication*) offers the following functionality: (1) identify and authenticate a user, (2) let the user play a particular role, (3) let the user choose a particular part, and (4) change the part description in both of the local databases from which parts are derived.

After the big-bang (i.e. the process that represents IROApplication[1] is loaded into memory) a default *system* object immediately passes control to IROApplication[1] in sending the message *main*. During the execution of method *main*, the application first opens the IRO-DB database "example" and tries to *login* a user, i.e. it identifies User[7] and authenticates him/her using password "abc". The method *authenticate* actively changes the activity stack in that it lets an authenticated user work *on-behalf* of the calling application. Next, the method *login* identifies that the user wants to *play* Role[2] and sends the corresponding message *play* to the particular role. The method *play* actively changes the activity stack again in that it lets the requested role work *instead-of* the active user. Third, the application identifies that the user wants to modify the description of PART[15] to the value "Part15" and sends the corresponding message *description*("Part15") to the particular part object which in turn sets the description attributes of both import objects S_PART[15] and I_PRT[15] from which PART[15] has been derived. After completing the task the application performs a *logout*, *closes* IRO-DB and *exits*, the corresponding process is terminated. The following section goes into detail for specifying authorizations within OOAC and illustrates some means to facilitate this process, for instance, using template or conditional authorizations.

² We assume the existence of exceptions in the object database here since they are the most intuitive way to handle errors.

Source Object (Activity Stack)	Target Object	Message (Method Call Stack)
system	IROApplication[1]	main(1,"1")
IROApplication[1]	IROApplication[1]	openDB("example")
IROApplication[1]	IROApplication[1]	login("7","abc","2")
IROApplication[1]	User[7]	authenticate("abc")
User[7]	User[7]	activate(OnBehalf)
User[7]	Role[2]	play
Role[2]	Role[2]	activate(InsteadOf)
Role[2]	PART[15]	description("Part15")
PART[15]	S_PART[15]	description.set("Part15")
PART[15]	I_PRT[15]	prt_desc.set("Part15")
Role[2]	IROApplication[1]	logout
IROApplication[1]	IROApplication[1]	closeDB
IROApplication[1]	IROApplication[1]	exit
system		

Table 1: Changes within the activity stack while executing an example application.

3.4 An Object Authorization Language (OAL)

Authorization in OOAC specifies the messages a source object may send to a target object. In order to describe authorizations in a declarative manner we propose an *object authorization language* (OAL). The authorizations necessary for the simple example application mentioned in the previous section can be expressed in OAL as follows:

```

ALLOW system SENDING main, exit TO
  IROApplication[1];
ALLOW IROApplication[1] SENDING
  authenticate TO User[7];
ALLOW User[7] SENDING play TO Role[2];
ALLOW Role[2] SENDING
  description(String) TO PART[15];
ALLOW PART[15] SENDING set TO
  S_PART[15].description,
  I_PRT[15].prt_desc;

```

The subsequent sections describe the 3 kinds of authorizations within OAL, namely *template*, *conditional*, and *negative* authorizations.

3.4.1 Template Authorization

The OAL contains mechanisms to relieve the administrative effort of specifying authorizations. The characters '*' (*any*) and '\$' (*arbitrary*) can be used to specify sets of objects and/or messages respectively to specify an arbitrary object or message that can be referred to from other points within an authorization. The latter concept is especially useful for conditional authorizations (see section 3.4.2).

Using the template characters '*' and '\$', an object, either source or target, may be specified in one of the ways shown in Figure 6:

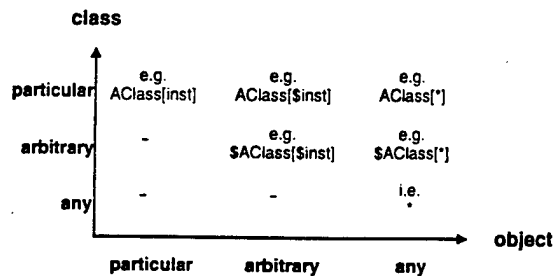


Figure 6: Template object specifications within OAL.

Both, the X and Y axis show the possible types of specifications which can be one of *particular* (by name), *arbitrary* (by character '\$' followed by an identifier) or *any* (by character '*'). The X axis corresponds to object definitions, the Y axis to class definitions. Thus, a complete object specification may be a *particular object* of a *particular class* (AClass[inst]), an *arbitrary object* of a *particular class* (AClass[\$inst]), *any object* of a *particular class* (AClass[*]), an *arbitrary object* of an *arbitrary class* (\$AClass[\$inst]), *any object* of an *arbitrary class* (\$AClass[*]), and *any object* of *any class* (*).

The OAL also allows to specify template messages. Thus, a complete message specification may be a *particular message* (e.g. \$msg1(AClass[*],*), parameters may optionally be specified as a set of object specifications placed within parenthesis and separated with commas), an *arbitrary message* (\$aMessage) of the object's interface, and *any message* (*) of the object's interface.

The example authorizations described in the previous sub-section could be made more general in using template authorizations like the following:

```
(1) ALLOW system SENDING main, exit TO
    Application[*];
(2) ALLOW Application[*] SENDING
    authenticate TO User[*];
(3) ALLOW User[$u] SENDING play TO
    User[$u].roles[*];
(4) ALLOW Role[2] SENDING
    description(String) TO PART[*];
(5) ALLOW PART[$p] SENDING * TO
    PART[$p].origin[*];
```

The set of authorizations specifies that the default *system* object may send messages *main* and *exit* to any

application (1) which in turn may authenticate any of the known users (2). Each particular user may *play* those roles (s)he is a member of (3). Role[2] may change the *description* of any PART object (4), and a particular PART object may send any message to those objects the PART object has been derived from, i.e. the *origin* of the derived object (5).

3.4.2 Conditional Authorization

In many cases it is desirable to specify constraints on authorizations, for instance, dependencies on attribute values, time, location, or even on other authorizations (like within the concept of implied authorization, see for instance [3]). OOAC allow to specify two kinds of conditional authorizations:

- dependent on *object state*, and
- dependent on *other authorizations*.

Suppose the following:

```
IF Date.now() < User[$u].expirationDate()
THEN
    ALLOW User[$u] SENDING play TO
        Role[Subscriber];
END
```

In this case, the user objects are designed to have an attribute holding an expiration date. The message *expirationDate* returns this date which is then compared to the current date (message *now* is a class member of *Date* returning the current date). The authorization specifies that any user of the database is allowed to *play* Role[Subscriber] if the user's validation date has not expired.

The authorization below is dependent on another authorization and realizes a mutual exclusion constraint in that Role[2] is denied for what Role[1] is authorized to.

```
IF ALLOWED Role[1] SENDING $m TO
$AClass[$inst] THEN
    DENY Role[2] SENDING $m TO
    $AClass[$inst];
END
```

Summarizing, the OAL provides possibilities to specify authorizations that depend on object state and/or other authorizations. The former allows to realize any kind of value dependency which is possible using the database objects and messages

(including time, location, etc.). The latter can be used to realize particular policies e.g. for the concept of implied authorization or for mutually exclusive roles.

3.4.3 Negative Authorization

Usually, authorization models allow to specify positive authorizations (permissions) based on a *closed world* assumption saying that any access is denied unless explicitly permitted. Authorization models have been enriched with the concept of negative authorizations (prohibitions) for more flexibility combined with an *open world* assumption (any access is allowed unless explicitly prohibited). A mixture of positive as well as negative authorizations is feasible enabling to specify exceptions from general specifications.

Within OOAC, authorization may be either based on an *open* or a *closed* world assumption. Furthermore, an access may be either *allowed*, or *denied*, respectively, a previously allowed/denied access may be *revoked*. The sequence of authorizations is relevant with respect to conflicts either due to template authorizations or due to coexistence of permissions and prohibitions. Consider the following sequence of authorizations:

```
(1) ALLOW User[*] SENDING
    description() TO PART[*];
(2) DENY User[47] SENDING
    description() TO PART[*];
(3) DENY User[*] SENDING
    description(String) TO PART[*];
(4) ALLOW User[11] SENDING
    description(String) TO PART[*];
```

The sequence specifies that principally any user is *allowed* to retrieve the description of PART objects (1) except for User[47] (2). On the other hand, any user is principally *denied* to change the description of PART objects (3) with the exception of User[11].

3.5 Authorization and Access Control

A set of policies characterizes the basic attitudes of OOAC. We try to minimize the number of policies in order to emphasize the flexibility offered by *template* and *conditional* authorizations. Nevertheless, some fundamental policies are specified in this sub-section

which are useful and are intended to reflect the principle nature of object orientation.

3.5.1 Fundamental Policies

The first policy has already been mentioned before. It reduces the number of access controls within OOAC in providing a base class *NamedObject* which has to be sub-classed for objects that could be named and should be an issue for access controls (i.e. protection objects).

(protection-object): access controls are applied to messages sent to named objects.

In case that a non-protection object initiates a message that object has to work *on-behalf* of a protection object in order to be possibly authorized for the particular message. Any message that is sent to a non-protection object can only be controlled at the programming language level.

The second fundamental policy serves the principle of encapsulation in that it lets any object use its own interface, freely.

(object-interface): any object is allowed to send any possible message to itself.

This policy can also be disabled if desired in using the following template authorization: "DENY \$AClass[\$inst] SENDING *". The negative authorization does not specify a target object which is then supposed to be equal to the source object. It says that an arbitrary object of an arbitrary class must not send messages (defined for that class or inherited, see inheritance 1 and 2 policies below) to itself.

3.5.2 Object/Class Methods

A method may either be defined for individual objects or for all objects of a class at once. *Object methods* are the regular case and can be executed independently on any of the class' objects. Authorizations for object methods may include an *instantiation clause* ("[...]") and thus be specified for individual objects. A *class method* can only be executed on all objects of a class at once, not independently on a particular object of a class. An

example for a class method taken from the basic schema illustrated in

Figure 2 is *NamedObject.lookup* which takes a name and returns the corresponding object from within the extent of the regarded class, e.g. *User.lookup("user1")* returns the instance of class *User* that is named "user1" (the expression is equal to *User[user1]* within the OAL). Consequently, an authorization for a class method does not allow to specify an *instantiation clause* for the target object within the OAL. However, not all of the object systems distinguish between object- and class-methods.

3.5.3 Overloading

Overloaded methods have the same name but different signatures within one class. Examples for overloaded methods are *name()* and *name(String)* of class *NamedObject* again taken from the basic schema. The former returns the name of an object, the latter allows to change the name of an object. Overloaded methods are distinguished using their signature, i.e. the number and types of parameters as well as the type of an optionally returned result. Note, that most object systems ignore the result type due to technical issues.

3.5.4 Inheritance

Methods may be inherited, that is, specified and implemented in a direct or indirect super-class of the regarded class. With respect to access controls, all methods that are inherited by an object are regarded as components of the object's interface which leads to the following two policies:

(inheritance 1): authorizations may be specified for messages that are inherited from super-classes of the target object's class.

Thus, an authorization like "ALLOW User[1] SENDING name() TO DerivedObject[*]" could be specified to allow User[1] to retrieve the name of derived objects, for instance, although the particular method is defined and implemented in class *NamedObject*.

(inheritance 2): an authorization to send *any* message defined for object *o* includes those messages that are inherited from any of the super-classes of *o*.

For instance, the authorization "ALLOW User[1] SENDING * TO PART[1]" allows User[1] to send any message defined for class *PART* (i.e. *description*, and *quantity*) as well as any message inherited from the super-classes (i.e. *name*, *lookup*, *new*, *copy*, and *delete*) to PART[1].

We do not want to go into detail for multiple inheritance since this is an optional feature of object databases (compare [1]). Nevertheless, if an object system supports multiple inheritance, OOAC has to deal with the possibility of ambiguous messages (i.e. messages that cannot clearly define which method of the super-classes has to be executed). Furthermore, an object may combine methods inherited from both, protection objects and non-protection objects, which requires to additionally control some messages sent to non-protection objects.

3.5.5 Polymorphism

Inherited methods may be *overridden*, i.e. the implementation of the method may be changed or extended without changing the name and signature of the method. The method that should be executed can be determined by examining the *type* of the regarded object. If the decision depends on the *dynamic* type of an object the concept is also called *late binding*, since the dynamic type can only be determined at run-time. The following two policies dealing with the polymorphism property of objects can thus be stated:

(polymorphism 1): an authorization to send message *m* to object *o* additionally holds for any possible form (polymorphism) of *o* that can respond to *m*.

For instance, the authorization "ALLOW User[1] SENDING name() TO PART[1]" additionally allows User[1] to send *name()* to PART[1] in the form of a *DerivedObject* or a *NamedObject* instance since these super-classes of *PART* can respond to message *name()*.

(polymorphism 2): an authorization to send message *m* to any object that is an instance of a particular class *C* include those objects that are instances of sub-classes of *C* which thus *could* be instances of *C* as well (due to polymorphism).

The authorization "ALLOW User[1] SENDING name() TO DerivedObject[*]" allows User [1] to send message *name()* to any object that is an instance of class *DerivedObject* (which in this case will be an empty set since *DerivedObject* is an abstract class). Additionally the message *name()* may be sent to those objects that *could* be instances of class *DerivedObject*, i.e. that are instances of one of the sub-classes of *DerivedObject*, namely *PART*.

3.5.6 Complex Objects

As mentioned above, complex objects are objects that are composed of simpler ones. In this paper, we assume an object system that completely encapsulates the attributes of an object, that is, the attributes can only be accessed by other objects in sending messages that can be controlled. Only the object itself is allowed to manipulate its attributes directly. Each attribute value in fact is a reference to an object. Some objects are collections and can be used to aggregate other objects (e.g. *Collection*, shown in

Figure 2). The concept is orthogonal in the sense that a collection may hold any object and thus may hold other collections, too.

In [12], we proposed some policies concerning implied authorization for components of complex objects, saying that if a subject (e.g. a user) is authorized to access a complex object, the subject should be implicitly authorized to access each component of the object and thus the complex object as a whole. OOAC do not use any automatic authorization mechanisms. Instead, each object has to be authorized for the actions it wants to execute which is orthogonal for subjects since they are themselves objects in OOAC. For retrieving the name of a derived PART object, for instance, the PART object has to be authorized to retrieve the name of any (import) object the PART has been derived from. An adequate authorization in OAL could be formulated as "ALLOW PART[\$p] SENDING name() TO

PART[\$p].origin[*]" authorizing an arbitrary PART object to retrieve the name of its origin objects.

Another problem is that some object systems provide types that are system defined (e.g. int, float, etc.) and do not represent first-class objects. Furthermore, attributes might be declared public, that is, they may be directly accessed by other objects without using the object's interface. Assume the attribute *description* of class *S_PART* to be a public *String*. Since strings are non-protection objects (see

Figure 2) the policy concerning protection objects has to be extended as follows:

(protection-object): access controls are applied to messages sent to protection objects and to those non-protection objects that are declared as public parts of a protection object.

The following authorization could be specified which allows User[1] to assign descriptions to any object of the origin of PART[1]:

ALLOW User[1] SENDING set TO
PART[1].origin[*].description;

System defined types that are declared as public attributes are handled by OOAC as quasi objects having the minimum interface *get* (the attribute value) and *set* (the attribute value) which allows to specify authorizations for these two messages.

4 Conclusions and Future Work

In this paper we introduced a new concept for access controls that is especially tailored for true object-oriented environments. The concept called *object-oriented access controls* (OOAC) is based upon the following assumptions: (1) everything within the object-oriented environment is regarded as an object, (2) thus, security subjects (e.g. users, roles, etc.) are regarded as first-class objects, too, (3) messages are the only means for communicating to other objects.

In consequence, OOAC deal with controlling the flow of messages among the objects of an object database. An *object authorization language* (OAL) has been proposed that allows to specify the set of

messages an object is allowed or denied to send to other objects in a declarative manner. The OAL provides means to specify *template* authorizations (relevant to a *set* of objects and/or messages), *conditional* authorizations (depending on object state or other authorizations) and *negative* authorizations (denying access rather than allowing it). Furthermore, we presented a minimal set of policies that corresponds to those properties commonly accepted to be inherent to object-oriented systems. The policies address some characteristics concerning the object interface, inheritance, polymorphism, and complex objects. On the other hand, OOAC do not impose a particular kind of access control policy (e.g. discretionary, role-based, or mandatory access controls). Instead, any known policy or even any policy developed in future may be implemented using OOAC since the structure and behavior of database objects exactly determine the ways an object may be accessed as well as the ways an object can be protected against unauthorized access. The feasibility of OOAC has been demonstrated in applying the concept to IRO-DB II, an extension of the database federation IRO-DB, which provides interoperable access between relational and object-oriented database systems within the world-wide-web.

Future research efforts will concentrate on implementing ownership-based (e.g. DAC), role-based (RBAC) or mandatory (MAC) access controls within OOAC. Furthermore, administration issues will be addressed assuming the existence of meta classes (like *Class*, *Attribute*, *Message*, *Method*, *Schema*, *Authorization*, etc.) as parts of the application schema allowing to apply OOAC for controlling schema modifications and/or security administration in the same way as controlling simple object access.

5 References

- [1] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S. (1989) The Object-Oriented Database System Manifesto. *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan.
- [2] Atwood, T., Duhl, J., Ferran, G., Loomis, M., and Wade, D. (1993) *The Object Database Standard: ODMG-93, Release 1.1*. Morgan Kaufmann Publishers, San Francisco, California.
- [3] Bertino, E., Kim, W., Rabitti, F. and Woelk, D. (1991) A Model of Authorization for Next-Generation Database Systems. *ACM ToDS*, Vol. 16/1.
- [4] Bertino, E., Martino, L. (1991) Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer*, April 1991, pp33-47.
- [5] Bertino, E. (1992) Data Hiding and Security in an Object-Oriented Database System. *Proc. 8th IEEE Int. Conf. on Data Engineering*, Phoenix, Arizona.
- [6] Bertino, E., Samarati, P. (1993) Research Issues in Discretionary Authorizations for Object Bases. *Workshop in Computing: B Thiraisingham, R. Sandhu, T.C. Ting (Eds.) Security for Object-Oriented Systems*, Washington DC.
- [7] Brüggemann, H. H., (1991) Rights in an object-oriented environment. *Proc. 5th IFIP 11.3 Working Conference on Database Security*. Shepherdstown, WV, 1991.
- [8] Busse, R., Fankhauser, P., Huck, G., Klas, W. (1994) IRO-DB An object-oriented approach towards federated and interoperable DBMS. *Proc. of the International Workshop on Advances in Databases and Information Systems (ADBIS'94)*, Moscow, Russia, Russian Academy of Sciences.
- [9] Busse R., Fankhauser P., Neuhold E.J. (1994a) Federated Schemata in ODMG, *Proc. 2nd Int. East/West Database Workshop*, Klagenfurt, Austria.
- [10] Castano, S., Fugini, M., Martella, G., Samarati, P. (1995) *Database Security*. Addison-Wesley.
- [11] Essmayr, W., Kastner, F., Pernul, G., Preishuber, S., and Tjoa, A. M. (1995) Access Controls for Federated Database Environments. *Proc. Joint IFIP TC 6 and TC 11 Working Conf. on Communications and Multimedia Security*, Graz, Austria.
- [12] Essmayr, W., Kastner, F., Pernul, G., Preishuber, S., and Tjoa, A. M. (1996) Authorization and Access Control in IRO-DB. *Proc. of the 12th Int. Conf. on Data Engineering*, New-Orleans, Louisiana, USA.

- [13] Essmayr, W., Kastner, F., Pernul, G., Tjoa, A M. (1996a) The Security Architecture of IRO-DB. *Proc. 12th IFIP Int. Conf. on Information Security*, Island of Samos, Greece.
- [14] Fernandez, E.B., Larrondo-Petrie, M.M., Gudes, E. (1993) A Method-Based Authorization Model for Object-Oriented Databases. *Workshop in Computing: B Thuraisingham, R. Sandhu, T.C. Ting (Eds.) Security for Object-Oriented Systems*, Washington DC.
- [15] Fernandez, E.B., Gudes, E. and Song, H. (1994) A Model for Evaluation and Administration of Security in Object-Oriented Databases. *IEEE Trans. on Knowl. & Data Eng.*, Vol. 6/2.
- [16] Gal-Oz, N., Gudes, E., Fernandez, E. B. (1993) A Model of Methods Access Authorization in Object-Oriented Databases. *Proc. 19th VLDB Conference*, Dublin, Irland.
- [17] Jonscher, D., Moffett, J. D., Dittrich, K. R. (1993). Complex subjects or: The Striving for Complexity is Ruling our World. *Proc. 7th IFIP 11.3 Working Conference on Database Security*. Huntsville, AL, 1993.
- [18] Pernul, G.(1994) Database Security. In: *Advances in Computers*, Vol.38, pp. 1-72. (M. C. Yovits, ed.). Academic Press.
- [19] Rumbaugh, J., Blaha, M., Premerlani, W.R., F. Eddy, Lorensen, W. (1991) *Object-Oriented Modeling and Design*, Prentice-Hall.
- [20] Sheth, A.P. and Larson, J.A. (1990) Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, Vol.22/3.

6 Acronyms

DAC	Discretionary Access Controls
IRO-DB	Interoperable Relational and Object-Oriented Databases
LDA	Local Database Adapter
MAC	Mandatory Access Controls
OAL	Object Authorization Language
ODMG	Object Database Management Group
OOAC	Object-Oriented Access Controls
RBAC	Role-Based Access Controls

Administration Policies in a Multipolicy Authorization System

E. Bertino E. Ferrari

Dipartimento di Scienze dell'Informazione
Università di Milano
20135 Milano, Italy

Abstract

This paper describes the administration policies supported by the MultiPolicy Authorization System (MPAS). Several administration policies are supported including centralized administration, decentralized administration with delegation and transfer, and joint administration. In the paper we first present an overview of the MPAS architecture. We then discuss the various administration policies and formalize some aspects of the proposed administration model.

1 Introduction

The introduction of an access control system within any organization entails two main tasks. The first task is the identification and specification of suitable *access control policies*. An access control policy establishes for each user (or group of users, or functional role within the organization) the actions the user can perform on which object (or set of objects) within the system under which circumstances. An example of access control policy [2, 3] is that "all programmers can modify the project files every working day except Friday afternoon". The second task is developing a suitable *access control mechanism* implementing the stated policies. Because of the richness of possible access control policies, this second task is quite difficult. Current access control mechanisms are tailored to few, specific policies and are unable to satisfactorily support access control requirements of several applications. In most cases, either the organization is forced to adopt the specific policy built-in into the access control mechanism at hand, or access control policies must be implemented as application programs. Both situations are clearly unacceptable. Many advanced applications, such as workflow applications, and computer-supported cooperative work, have articulate and rich access control requirements. Therefore, those applications cannot be adequately supported by a single-policy access control mechanism. Implementing access control policies as application programs, on the other hand, makes it very difficult to verify and

modify the access control policies and to provide any assurance that these policies are actually enforced.

A possible approach is to develop flexible access control mechanisms, able to support different access control policies for possible different objects within the system. We refer to such a system as a *multipolicy access control mechanism*. Consider the classical *closed* and *open* policies. Under the former, a subject can access an object only if an explicit *positive* authorization is specified. Under the latter, a subject can access an object only if there is no explicit denial (also called *negative authorization*). It is easy to encounter many situations where some objects within a system must be governed by the open policy (such as data objects containing information available to the majority of users), whereas other objects within the same system need a stricter control (such as data objects containing information available to few, selected users), thus requiring a closed policy.

Several proposals in the areas of database systems and operating systems have addressed issues related to multipolicy access control mechanisms. Proposals in the database area include the flexible authorization model proposed in [5], and the Chassis system [11] specifically addressing access control for federated heterogeneous database systems. Proposals in the operating systems area include Trusted Mach [7] and DTOS [9].

The flexible authorization model presented in [5], developed as an extension of the Orion authorization system [12] and recently formalized in a logical framework [6], supports both positive and negative authorizations. It, moreover, supports exceptions and different conflict resolution policies. The Chassis system provides different local security monitors, each implementing the specific policy of a site in the federation. The local monitors are complemented by a global authorization layer, supporting global authorization policies. The local authorization policies can be quite different. A relevant goal of the Chassis project is how to handle authorization conflicts among different site

and how to map global authorizations onto local ones.

All the above proposals, however, do not provide sophisticated authorization administration policies. Authorization administration refers to the function of granting and revoking authorizations. It is the function by which authorizations are entered (removed) into (from) the access control mechanism. In most of the above approaches, the administration policies are either based on the centralized approach, or on the ownership approach possibly complemented with the administration delegation approach. Moreover, no multiple administrative policies within the same system are supported.

Two different administration policies are briefly discussed by Fernandez, Gudes and Song [8] in the framework of an access control mechanism for object-oriented database systems. The first policy is based on the ownership approach. The users own the data and administer their data. Under the second policy, some special users (administrators) control the access to data. However, their access control model supports only the second policy. By contrast, our model supports both those policies, and many others, so that users may choose the policy which best suits their application requirements.

Finally, we would like to mention the brief preliminary discussion reported in [15] addressing the use of delegation and joint actions in authorization systems. These mechanisms are able to support sophisticated authorization schemes, particularly useful when dealing with complex information systems and distributed systems. Even though the authors do not address authorization administration, we believe that many issues pointed out in the discussion are relevant to our approach.

In this paper we present the authorization administration facilities provided as part of the MultiPolicy Authorization System (MPAS) being currently developed at the University of Milano. MPAS supports both the specification and implementation of multiple authorization and administration policies by, at the same time, clearly separating specification from implementation. The system supports a large variety of administration policies from centralized administration, either DBA¹ or owner-based, to joint-based administration, by which several users are jointly responsible for authorization administration. MPAS currently supports only discretionary access control policies. The reason is that the applications using discretionary policies are those that usually require high flexibility. We plan, however, to explore other types of

access control policies, such as the chinese wall policy.

The remainder of this paper is organized as follows. Section 2 briefly discusses the MPAS architecture. Section 3 presents the various administrative policies supported by MPAS. Section 4 presents a formalization of some aspects of our authorization administration model. Finally, Section 5 concludes the paper and outlines future work.

2 Architecture of MPAS

The architecture of MPAS is illustrated in Figure 1. The system consists of two main environments: the *policy specification environment* and the *run-time environment*. In discussing the architecture, we will cast the discussion in terms of database systems, by assuming that the items to be protected are data objects, such as relations in a relational DBMS, or objects in an object DBMS. However, we believe that the discussion is also valid in other contexts.

2.1 Policy specification environment

The policy specification environment supports all functions concerning policy specification for object authorization and authorization administration.

Example 1 Consider a table `Public.info` containing information available to all employees of a given company. An example of authorization policy is that access to table `Public.info` must be governed by the open policy [5], whereas an example of administration policy is that authorizations on `Public.info` can only be granted by a DBA (in practice, this means that only the DBA can issue access denials).

A number of predefined authorization policies are supported (denoted in the reference architecture as the authorization policies library), including the following: traditional closed and open policies; the closed policy with negation and the conflict resolution principle based on "denials take precedence"; the closed policy with negation and the conflict resolution principle based on "the most specific authorization takes precedence" [5]. In particular, the last two policies have conflict resolution principles for dealing with conflicting authorizations² granted on the same object to the same subject. It is also possible for the policy officer to specify custom-made policies by using a special purpose language, based on rules [4].

A number of predefined administration policies are supported (denoted in the reference architecture as administration policies library) that will be discussed in the next section. All the specifications are checked

¹DBA stands for database administrator.

²Two authorizations conflict if one is a positive authorization whereas the other is a negative one.

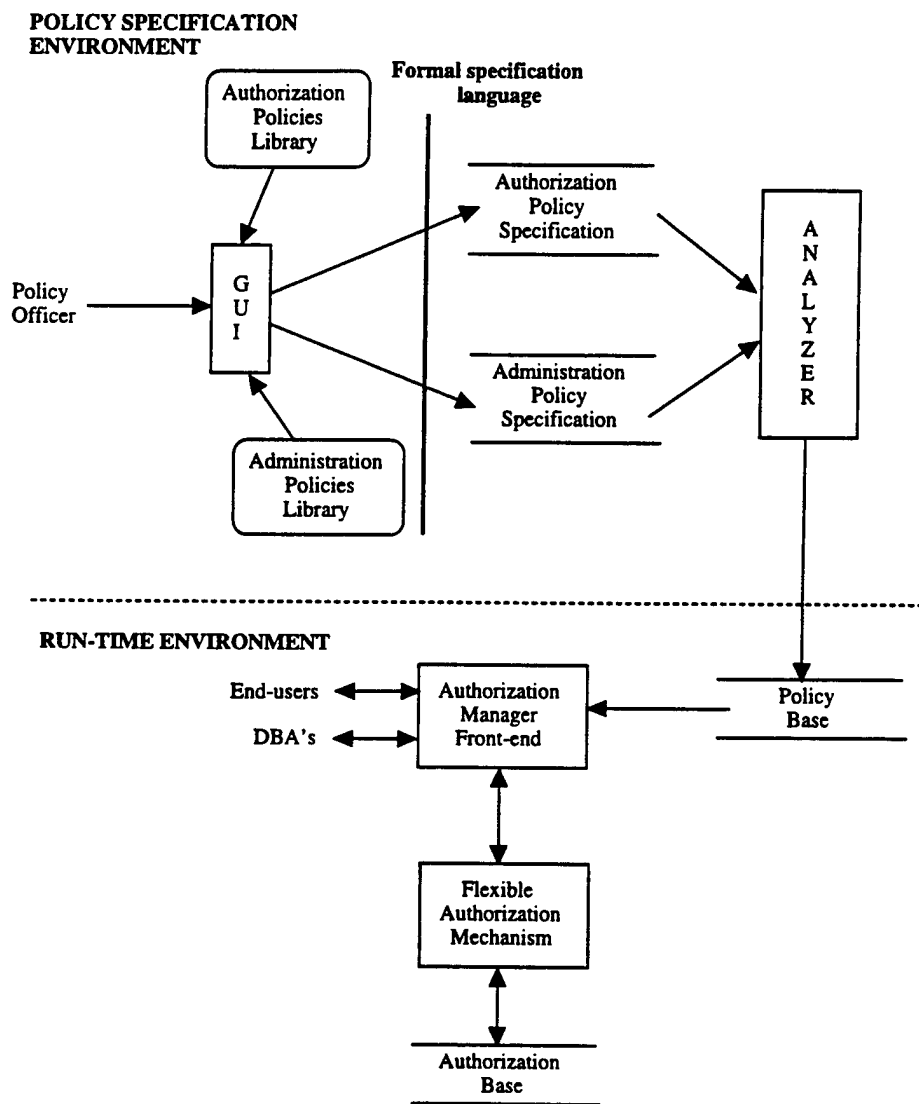


Figure 1: Architecture of the multipolicy authorization system (MPAS)

for consistency and correctness by the analyzer [4]. The results of the analysis is a policy base encoding the administration and authorization policies for each data object.

An important aspect of our system is that *parametric policies* can be also specified. A parametric policy is parametric with respect to the objects to which it applies. It allows to specify policies such as "All tables created by the role Secretary must be authorized under the open policy" or "All documents pertaining to project MPAS must be authorized under the closed policy". Parametric policies basically contain conditions stated against *authorization attributes* of data objects. Authorization attributes contain information about the administered objects that is relevant from security point of view. They compose the *security profile* of the data object. Each subject also has a similar security profile. Parametric policies are very important for systems with large numbers of subjects and objects to be protected. In such systems, specifying policies for each object would not be viable.

2.2 Run-time environment

The run-time environment consists of a flexible authorization mechanism which is the core of MPAS, and of a front-end. The flexible authorization mechanism implements a generalized authorization model able to support a large number of policies [5]. The front-end has the task of verifying the user's and DBA's request with respect to the policies stated by the policy officer (stored in the policy base) and mapping them onto the generalized authorization model.

Example 2 Consider table `Public_info` from Example 1 for which an open authorization policy and a DBA administration policy have been specified. Suppose that a DBA enters a positive authorization for a certain user on this table. Such authorization is rejected by the front-end because only negative authorizations can be specified for this table, according to the open policy.

3 Administration Policies

In this section we discuss the predefined administration policies supported by our system. Note, however, that additional policies can be specified by using the rule-based language supported by the policy specification environment.

Before discussing the policies it is important to recall that authorization administration consists of issuing grant and revoke requests to the authorization system. Those requests, that must be consistent with the administration policies, enter or remove authorizations from the authorization base (cfr. Figure 1).

They are issued by users that must be, in turn, properly authorized. We will also refer to the notion of *object creator*; it is the user who has created the object or on behalf of whom the object has been created (for example, within an application program run by the user).³

The following administration policies are supported by MPAS:

- **DBA administration:** under this policy, only the DBA can issue grant and revoke requests on a given object.⁴ This policy is highly centralized (even though different DBAs can manage different parts of the database) and it is seldom used in current DBMSs, but in the simplest systems.
- **Object "curator" administration:** under this policy, a user, not necessarily the creator of the object, is named administrator of the object. Under such policy, even the object creator must be explicitly authorized to access the object.
- **Object owner administration:** under this policy, which is commonly adopted by DBMSs and operating systems, the creator of the object is the owner of the object and is the only one authorized to administer the object.

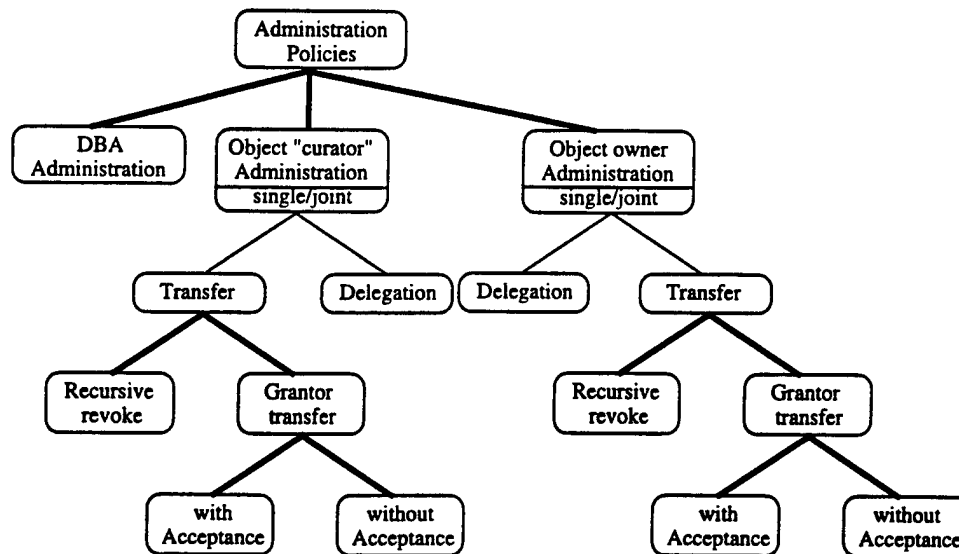
The second and third administration policies above can be further combined with *administration delegation* and *administration transfer*. Even though delegation and transfer could be applied to DBA administration, we do not have included this possibility because it is not very significant.⁵

By administration delegation we mean that the administrator of an object (either the owner or the curator) can delegate other users administration functions on the object. Delegation can be specified for selected access modes, for example only for read operations. In most cases, delegation of administration to another user implies also granting this user the privilege of accessing the object according to the same access mode specified in the delegation. Most current DBMSs support the administration policy based on the owner administration with delegation. Note that under the delegation approach, the initial administrator of the object does not lose his/her privilege to administer

³In some systems, a DBA can create an object on behalf of some users.

⁴Note that DBAs have also the authorization to give users the privilege to connect to the DBMS, and can also read and write all data objects created by other users.

⁵It can be simply implemented in terms of the other two policies.



Legend:

bold lines denote mutually exclusive administration options for the same data object

non-bold lines denote non-mutually exclusive administration options for the same data object

Figure 2: Taxonomy of the administration policies supported by MPAS

the object. Therefore, authorizations on the same object can be granted by different administrators. With respect to revoke operations, we take the approach, common to most systems, that only the *grantor* of an authorization can revoke it.

Administration transfer, like delegation, has the effect of giving another user the right to administer a given object. However, the original administrator loses his administration authorization, whereas under delegation the original administrator keeps his administration right. When transfer is used with owner administration it has the semantics of owner transfer; we call it *ownership transfer*. Therefore, the owner of the object is actually replaced by the user who has been delegated the administration. Transfer is very relevant in workflow applications, where objects (such as documents, and office forms) migrate among different departments (or organizational units) within the same organization. Often, because of those transfers, objects may enter different administration domains and it is, thus, important that the privileges of administering the objects be properly modified. Also, security requirements may dictate owner transfer. Consider a document which is initialized by a secretary and later on transferred to her boss who enters reserved information. In such a case, it is important that the initial

owner, i.e., the secretary, is no longer authorized to administer the object.

When dealing with transfer an important question concerns the authorizations granted by the former administrator (such problem does not arise in delegation because the former administrator retains his right to administer the object). The following two approaches are supported by MPAS for dealing with authorizations granted by the former administrator:

1. *Recursive revoke*: all authorizations granted by the former administrator are recursively revoked.
2. *Grantor transfer*: all authorizations granted by the former administrator are kept; however, the new administrator replaces the old one as grantor of the authorizations (and is thus able to revoke them). Note that the grantor transfer is not recursive. Therefore, if the older administrator has delegated other users for administration, those grants are left in place. Only, the new administrator becomes their grantor.

We provide a further transfer option. Transfer can be *with acceptance* or *without acceptance*. By acceptance we mean that the user to whom the administration (or ownership) is transferred must explicitly ac-

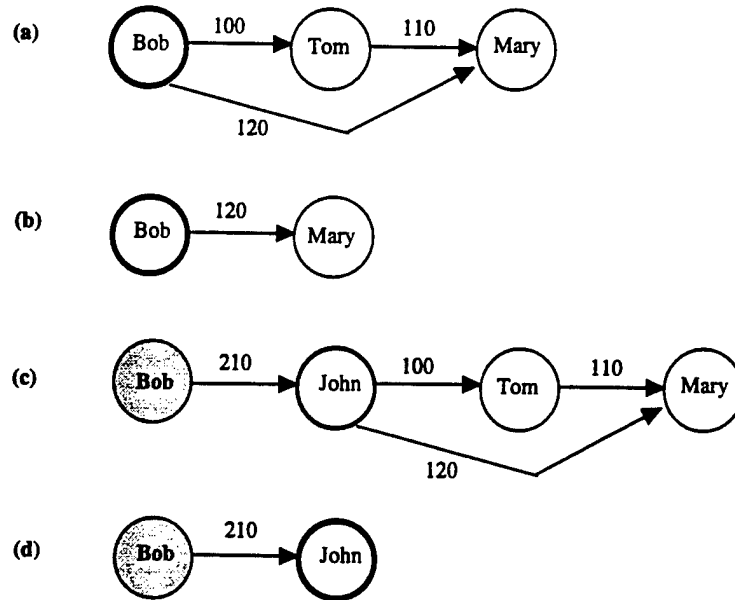


Figure 3: (a) An example of administration graph; (b) the graph after Bob revokes the delegation to Tom; (c) the graph after Bob transfers the ownership to John with grantor transfer; (d) the graph after Bob transfers the ownership to John with recursive revoke

cept the administration responsibility. Transfer without acceptance means that such explicit acceptance is not required. Explicit acceptance is important especially when the object ownership is transferred. Since the object owner is ultimately responsible for the object and may be liable for the content of the object, the explicit acceptance avoids that a user be transferred the ownership of an object, without even knowing about it.

Our model also supports *joint administration* of data objects. Joint administration means that several users are jointly responsible for administering the object. Joint administration can be used in both the object "curator" administration and object owner administration policy. Joint administration is particularly useful in computer-supported cooperative work (CSCW) applications where typically users cooperate to produce a complex data object (such as a document, a book, a piece of software, a VLSI circuit). In such applications, each user in the work group is responsible for producing a component of the complex object; therefore, no single user is the owner of the entire complex object. Authorization for a user to access a data object, administered under the joint administration policy, requires that all the administrators of the object issue a grant request. As a further option, MPAS supports joint administration with *quo-*

rum. Under such option, an authorization is granted to a user, if a number of administrators equal to the quorum have issued the proper grant request.

It is important to note that joint administration can be combined with the various options illustrated before. For example, one of the administrators of a data object may transfer or delegate its administration right to another user. A large spectrum of administration policies is thus obtained.

Figure 2 summarizes the administration policies supported by MPAS. In what follows we give some examples of the various administration policies supported by our model.

In the examples, we represent a sequence of delegation operations as a graph, called **administration graph**. In such a graph, a boldface node denotes the owner of an object, whereas a non-boldface node denotes a user who has been delegated the administration right on the object. There is an arc from node i to node j if the user represented by node i has delegated the administration right to the user represented by node j ; each arc is furthermore labeled with the delegation time. An example of administration graph is illustrated in Figure 3(a).

Moreover, sequences of grant requests for a given access mode m on a given object o are represented by means of **authorization graphs**. Nodes represent

users. There is an arc from node i to node j if the user represented by node i has granted an authorization for m on o to the user represented by node j . The arc is labeled with the time of the grant and with the granted access mode. Joint authorizations are represented by a special place-holder denoted by a vertical boldface bar. An example of authorization graph is shown in Figure 4(a).

The following example illustrates some of the administration options illustrated above for the case of single (i.e. non-joint) administration.

Example 3 Consider a table T which is administered under the owner administration policy with both delegation and transfer options. Suppose that Bob is the owner of the table. Under the above options, Bob can delegate other users the right to administer the object. Suppose that he grants such right to Tom at time 100⁶ and to Mary at time 120. Suppose, moreover, that Tom in turn delegates Mary the administration right at time 110. Figure 3(a) illustrates the corresponding administration graph.

Consider now the following cases:

1. Bob revokes the administration right from Tom. As a consequence, Mary loses the right received from Tom. However, she keeps the right received directly from Bob. Figure 3(b) shows the resulting graph.
2. Bob transfers at time 210 the ownership to John. Suppose that the transfer policy has been specified with the grantor transfer option. As illustrated by the graph in Figure 3(c), both Tom and Mary keep their administration rights. The graph includes a third type of node, called shadow node, which is used to keep track of previous object owners.
3. Bob transfers at time 210 the ownership to John. Suppose that the transfer policy has been specified with the recursive revoke option. As illustrated by the graph in Figure 3(d), both Tom and Mary lose their administration rights. As in the previous case, information about the former owner is kept in the graph.

The following examples illustrates joint administration.

Example 4 Consider a table T , administered under the joint administration policy. Suppose that Bob and Ken are the owners of the table. Suppose that the following grant operations are performed:

1. Bob grants Laura the Read access on T at time 105.
2. Ken grants Laura the Read access on T at time 110.

Figure 4(a) shows the resulting authorization graph.

Consider the following access requests issued by Laura:

1. Read access to T at time 105; such access is denied because, at time 105, Laura does not have the authorization from Ken.
2. Read access to T at time 115; such access is allowed because, at time 115, Laura possesses authorizations from both Bob and Ken.

The following example illustrates the difference between joint administration and delegation.

Example 5 Consider again table T . Suppose that it is administered under a non-joint policy and that Bob is its owner. Moreover, suppose that Bob delegates Ken the administration right at time 80. Suppose now that Bob and Ken perform the same grant operations illustrated in Example 4. Figure 4(b) shows the corresponding administration and authorization graphs. Consider again the access requests, listed in Example 4, issued by Laura:

1. Read access to T at time 105; such access is allowed, because Laura possesses the authorization granted from Bob.
2. Read access to T at time 115; such access is allowed because Laura possesses two authorizations, one from Bob and another from Ken.

Delegation and transfer also differ with respect to the semantics of the revoke operations [4].

Several interesting issues are related to the semantics of the delegation option when combined with joint administration. Indeed, one of the administrators of the object may delegate other users the administration of the object. This means that the delegated user may issue a grant request instead of the original object administrator. This grant request must be combined with the grant requests from all the other administrators (or from users delegated by them) before the authorization can actually be issued. If a quorum option is used, multiple grant requests from an administrator and his delegates amount to a single request with respect to the quorum computation. Such approach avoids that an administrator may reach the

⁶ We use here a simplified representation of time as an integer number. In real implementations, the system timestamp is used.

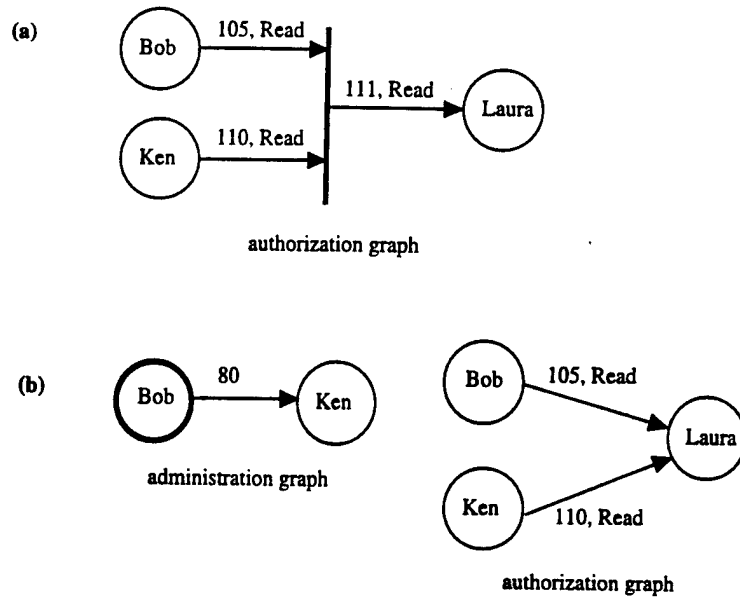


Figure 4: (a) The authorization graph for Example 4; (b) The administration and authorization graphs for Example 5

quorum by simply delegating other users the administration rights. The following examples illustrates joint administration with delegation.

Example 6 Consider a table T , administered under the joint ownership administration policy with quorum option. Suppose that Bob, Ken and George are the owners of the table. Suppose that the quorum is 2. Suppose that the following delegation and grant operations are issued:

1. Bob delegates John the administration authorization on T at time 100;
2. Bob grants Laura the Read access on T at time 120;
3. John grants Laura the Read access on T at time 130;
4. Ken grants Laura the Read access on T at time 150.

Figure 5 shows the resulting administration and authorization graphs.

Consider the following access requests issued by Laura:

1. Read access to T at time 130; such access is not allowed, because two read authorizations have been issued for Laura, but they are from an administrator and a delegate of him;

2. Read access to T at time 160; such access is allowed because Laura now possesses two authorizations, granted by two different administrators (or their delegates).

Finally, note that even if Bob had not granted the Read authorization to Laura at time 120, she would still be able to access T at time 160 (provided that authorizations (3) and (4) above had been granted).

We refer the reader to [4] for a discussion and for a formal definition of the semantics of delegation when combined with joint administration.

4 Formal Model

In this section we formalize some aspects of our authorization administration model.

Let O be the set of objects and U the set of users in the system. Let DBA denotes the set of all users having DBA authorizations. Let \mathcal{N} denotes the set of natural numbers. Moreover, let $\mathcal{PS} = \{DBA, \text{object-curator}, \text{object-owner}, \text{joint-object-curator}, \text{joint-object-owner}\}$ denotes the set of administration policy types. An authorization administration policy is defined as follows.

Definition 1 (Administration policy) An administration policy is a 7-tuple $[o, pt, delegation_opt, transfer_opt, acceptance_opt, revoke_opt, vote_opt]$.

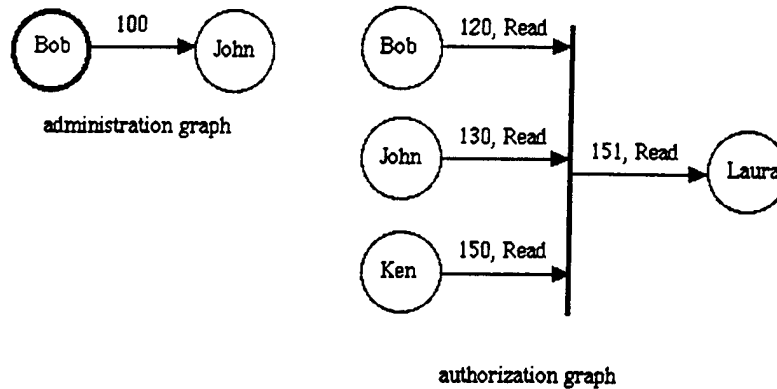


Figure 5: Administration and authorization graphs for Example 6

opt], where $o \in O$, $pt \in \mathcal{PS}$, $delegation_opt \in \{delegation, no-delegation, nil\}$, $transfer_opt \in \{transfer, no-transfer, nil\}$, $acceptance_opt \in \{acceptance, no-acceptance, nil\}$, $revoke_opt \in \{revoke, grantor-transfer, nil\}$, $vote_opt \in \{quorum, totality, nil\}$.

In the above definition, a number of components of the 7-tuple defining an administration policy are actually flags indicating specific options. Also, those flags may take a nil value. The nil value simply denotes that the flag is not significant for the specific policy type. Finally, the $vote_opt$ component is only significant for joint administration policy (for non-joint administration policies it always takes the nil value). It takes the *totality* value to denote that for an authorization to be granted all the administrators must have issued the proper grant requests; it takes the value *quorum* otherwise.

Example 7 The policy specifications below illustrate the above definition:

- The policy specification:

[Public.info, DBA, nil, nil, nil, nil, nil]

states that table *Public.info* must be administered by the DBA. All other components are not significant for this policy and are thus set to nil.

- The policy specification:

[T, object-owner, delegation, transfer, no-acceptance, grantor-transfer, nil]

states that table *T* must be administered by the owner. Moreover, both delegation and transfer are allowed on this table. Transfer is without acceptance and all granted administration authorizations are not revoked if the ownership is transferred. This policy is the one exemplified in case 2 of Example 3. Note that the $vote_opt$ is nil since the policy type is not a joint one.

- The policy specification:

[T, joint-object-owner, delegation, transfer, no-acceptance, grantor-transfer, totality]

states that table *T* is under a joint administration policy. In this case the $vote_opt$ component indicates that all the administrators must issue grant requests for an authorization to be actually granted to a user.

The policy administration base is a set of administration policies, denoted as \mathcal{PAB} .

Information specified by the administration policies are complemented with information about:

- which users are DBA;
- which administrators have delegated (transferred) to which other users administration authorizations;
- for each data object, its owner (owners), if the object is administered under the (joint) owner administration policy, or its "curator" ("curators") if the object is administered under the (joint) "curator" administration policy;

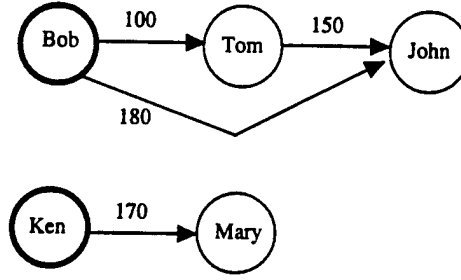


Figure 6: Administration graph with non-independent and independent administrators

- (iv) for each object, administered under a joint administration policy with quorum, the actual quorum value to use.

The above information is stored as facts into the policy base (see Figure 1).

Definition 2 (Delegation and transfer specification) A delegation (transfer) specification is a triple $[\text{grantor}, \text{grantee}, o]$, with $\text{grantor}, \text{grantee} \in U$, and $o \in O$.

The delegation (transfer) base is a set of delegation (transfer) specifications, denoted as $\mathcal{DL} (TR)$.

Definition 3 (Owner and curator specification) An owner (curator) specification is a pair $[o, u]$, with $o \in O$ and $u \in U$; u is referred to as an initial administrator of o .

Note that when the joint administration policy is used for an object, there are several initial administrators for the object.

The owner (curator) base is a set of owner (curator) specifications, denoted as $OWB (CB)$.

Definition 4 (Quorum specification) A quorum specification is pair $[o, n]$, with $o \in O$, $n \in \mathcal{N}$.

The quorum base is a set of quorum specifications, denoted as (QB) .

The following definition introduces the notion of delegates and states how the delegates of a given administrator are determined from the delegation base. Note that an administrator can have indirect delegates; indeed, an administrator may delegate a user the administration right and this user may, in turn, delegate other users.

Definition 5 (Delegates) Let u_i and u_j be users. Let o be an object. We say that u_j is a delegate of u_i for the administration of o if predicate $d(u_i, u_j, o)$, defined below, is True.

- $d(u_i, u_j, o) = \text{True}$, if $[u_i, u_j, o] \in \mathcal{DL}$;
- $d(u_i, u_j, o) = \text{True}$, if there exists $u_k \in U$ such that $d(u_i, u_k, o) = \text{True}$ and $d(u_k, u_j, o) = \text{True}$.

Let o be an object and let u be a user such that u is an initial administrator of o . The set $\mathcal{A}_u = \{u\} \cup \{u_i \mid u_i \in U \text{ and } d(u, u_i, o) = \text{True}\}$ denotes a set including u and all the delegates of u .

The following definition states the notion of independent administrators. It is the basis of the rule determining when a grant becomes valid in the case of joint administration.

Definition 6 (Independent administrators) Let u_i and u_j be users. Let o be an object. We say that u_i and u_j are independent administrators of o , if the following conditions are verified:

- $\exists u'_i$ initial administrator of o such that $u_i \in \mathcal{A}_{u'_i}$;
- $\exists u'_j$ initial administrator of o such that $u_j \in \mathcal{A}_{u'_j}$;
- $\nexists u_k$ initial administrator of o such that $u_i \in \mathcal{A}_{u_k}$ and $u_j \in \mathcal{A}_{u_k}$.

u_i and u_j are called non-independent administrators, otherwise.

The above definition states that two users, who have been delegated by other users to administrate a given object are independent if they have received their administration privilege from users that are, in turn, independent. Note that the initial administrators of the object are always independent.

Example 8 Consider the administration graphs in Figure 6. $\mathcal{A}_{\text{Bob}} = \{\text{Bob}, \text{Tom}, \text{John}\}$ and $\mathcal{A}_{\text{Ken}} = \{\text{Ken}, \text{Mary}\}$. The pairs of independent administrators include: (Bob, Ken), (Bob, Mary), and (Tom, Mary). The pairs of non-independent administrators include: (Tom, John), (Bob, John).

Function $grt(u, o)$

$grt : U \times O \rightarrow \{True, False\}$

let pt be the policy type of object o (determined from PAB)

case (pt):

DBA: if $u \in DBA$, then *True*, else *False*;

object-curator: if $[o, u] \in CB$ or

$(\exists u' \text{ such that } [o, u'] \in CB \text{ and } d(u', u, o) = True)$ then *True*, else *False*;

object-owner: if $[o, u] \in OWB$ or

$(\exists u' \text{ such that } [o, u'] \in OWB \text{ and } d(u', u, o) = True)$ then *True*, else *False*;

joint-object-curator: if $[o, u] \in CB$ or

$(\exists u' \text{ such that } [o, u'] \in CB \text{ and } d(u', u, o) = True)$ then *True*, else *False*;

joint-object-owner: if $[o, u] \in OWB$ or

$(\exists u' \text{ such that } [o, u'] \in OWB \text{ and } d(u', u, o) = True)$ then *True*, else *False*.

Figure 7: High level specification of function grt

The following rule formally states when a set of authorizations granted by administrators of an object to a user actually enables the authorization for this user.

Joint administration rule

Let $GU = \{u_1, \dots, u_n\}$ be the set of users who have granted an authorization for the same privilege on object o to user u . Let v be the vote policy for object o . Let GU^* be the maximal subset of GU such that $\forall u_i, u_j \in GU^*, u_i$ and u_j are independent administrators of o .⁷ The granted authorization is enabled if the following conditions are verified:

- If $v = \text{totality}$: let adm be the number of initial administrators of object o . Then $card(GU^*)^8$ must be greater than or equal to adm .
- If $v = \text{quorum}$: let q be the quorum required for object o . Then $card(GU^*)$ must be greater than or equal to q .

Note that the above definition implies that when two users u_1 and u_2 give the same authorization to the same user, these authorizations are both effective to enable the authorization for the user only if u_1 and u_2 have obtained the administer privilege by two independent sources, that is, there does not exist a user which gave the administer privilege to both u_1 and u_2 .

An important function that must be defined as part of the model is that of checking whether a user, wishing to perform a grant operation, is authorized to do

⁷This implies that there does not exist $GU' \subseteq GU$ such that $\forall u_i, u_j \in GU', u_i$ and u_j are independent administrators of o , and $GU^* \subset GU'$.

⁸ $card(GU^*)$ denotes the cardinality of set GU^* .

so. Authorization to perform a grant operation on an object depends on the administration policy established for the object. Figure 7 reports a high level specification of function grt ; such function receives as arguments the user wishing to perform the grant and the object on which the grant is to be issued. It returns *True* if the user is authorized to issue the grant, it returns *False* otherwise.

5 Conclusions

In this paper we have presented an overview of the administration policies supported by the MPAS multipolicy authorization system. The system supports a variety of policies that are obtained by providing several options, such as delegation and administration. Joint administration policies are also supported. Many of these policies are useful in advanced applications, such as workflow systems and computer-supported cooperative work and, in general, cooperative applications.

A notable feature of MPAS is the notion of parametric authorization policies. Such authorization policies are very useful when dealing with data objects created through an application program and which need to be accessed from the same application program (such as temporary tables), or when dealing with a large number of users and data objects, as in most real applications.

The system is currently under implementation. A preliminary implementation of a flexible authorization mechanism, a core component of MPAS, has been completed. We are extending both our model and architecture to provide a more accurate modeling of roles, along the lines discussed by Sandhu in [14] and

[13], and to incorporate temporal authorization features [1]. A final research direction we plan to pursue is related to the use of multipolicy systems in heterogeneous systems [10].

References

- [1] E. Bertino, C. Bettini, E. Ferrari, P. Samarati, "A Temporal Access Control Mechanism for Database Systems", *IEEE Trans. on Knowledge and Data Engineering*, Vol.8, No. 1, pp.67-80, February 1996.
- [2] E. Bertino, C. Bettini, E. Ferrari, P. Samarati, "Supporting Periodic Authorizations and Temporal Reasoning in Database Access Control", *Proc. of the Twenty-second International Conference on Very Large Data Bases Conference (VLDB'96)*, Bombay (India), September 3-6, 1996, Morgan Kaufman Publishers.
- [3] E. Bertino, C. Bettini, E. Ferrari, P. Samarati, "Decentralized Administration for a Temporal Access Control Model" *Information Systems*, Vol. 22, No. 4, to appear.
- [4] E. Bertino, E. Ferrari, "A Multipolicy Access Control System - Authorization Model and Architecture", Technical Report, Dipartimento di Scienze dell'Informazione, Università di Milano, 1997.
- [5] E. Bertino, S. Jajodia, P. Samarati, "Supporting Multiple Access Control Policies in Database Systems", *Proc. of the IEEE Symposium on Research in Security and Privacy*, Oakland (Calif.), May 1996, IEEE Computer Society Press.
- [6] E. Bertino, S. Jajodia, P. Samarati, and V.S. Subrahmanian, "A Unified Framework for Enforcing Multiple Access Control Policies", *Proc. of the ACM-SIGMOD International Conference on Management of Data*, 1997.
- [7] M. Branstad, H. Tajalli, F. Mayer, and D. Dalva, "Access Mediation in a Message Passing Kernel", *Proc. of the IEEE Symposium on Research in Security and Privacy*, Oakland (Calif.), May 1989, IEEE Computer Society Press.
- [8] E. B. Fernandez, E. Gudes, H. Song, "A Model for Evaluation and Administration of Security in Object-Oriented Databases", *IEEE Trans. on Knowledge and Data Engineering*, Vol.6, No. 2, pp.275-292, April 1994.
- [9] T. Fine and S.E. Minear, "Assuring Distributed Trusted Mach", *Proc. of the IEEE Symposium on Research in Security and Privacy*, Oakland (Calif.), May 1993, IEEE Computer Society Press.
- [10] L. Gong, and X. Qian, "The Complexity and Composability of Secure Interoperation", *Proc. of the IEEE Symposium on Research in Security and Privacy*, Oakland (Calif.), May 1994, IEEE Computer Society Press.
- [11] D. Jonscher, and K. Dittrich, "An Approach for Building Secure Database Federations", *Proc. of the Twentieth International Conference on Very Large Data Bases Conference (VLDB'96)*, Santiago (Chile), September 12-15, 1994, Morgan Kaufman Publishers.
- [12] F. Rabitti, E. Bertino, W. Kim, D. Woelk, "A Model of Authorization for Next-Generation Database Systems", *ACM Trans. on Database Systems*, Vol. 16, No. 1, pp. 88-131, March 1991.
- [13] R. S. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based Access Control Models". *IEEE Computer*, Vol.29, No.2, pp.38-47, February 1996.
- [14] R. S. Sandhu, "Role Hierarchies and Constraints for Lattice-based Access Controls", *Proc. of 4th European Symposium on Research in Computer Security*, Rome (Italy), September 1996.
- [15] V. Varadharajan, and P. Allen, "Joint Actions Based Authorization Schemes", *Operating Systems Review*, Vol.30, No.3, pp.32-45, July 1996.

Invited Talk

Chair: Teresa Lunt

Security Issues in Integrated Storage

Jose A. Blakeley, Microsoft

Distributed and Federated Systems
Chair: T. C. Ting

Designing Security Agents for the DOK Federated System

Zahir Tari

Department of Computer Science
Royal Melbourne Institute of Technology
Bundoora East Campus, Plenty Rd.
VIC 3083 Australia
zahirt@cs.rmit.edu.au

Abstract

This paper addresses two main issues of the DOK system, that is the design of a framework for enforcing security policies and a secure architecture which implements such a framework. Federated security policies are expressed as logic-based expressions (called "aggregation constraints") specifying the different combinations of transactions that a user is not allowed to issue, either in single or multiple states of a federation.

- *To enable efficient monitoring of aggregation constraints, state transition graphs are generated to model the different sub-computations of the constraints. Two marking techniques, namely LMT (Linear Marking Technique) and ZMT (Zigzag Marking Technique), are proposed to detect violations of federated security policies.*
- *To enable an effective enforcement of security policies, we designed a secure DOK architecture using specialised agents: (i) coordination agents allow the coordination of different federated activities, (ii) task agents perform specific tasks of the federation, and finally (iii) database agents provide the required information for coordination and task agents.*

1 Motivation

Relatively few databases are accessible over the Internet. With today's technology one would like to encapsulate a database and make it available over the Internet. A client using such databases would browse an old census database, look-up references in an object-oriented database system, access descriptions and pictures over the Internet, or combine different information using NCSA Mosaic, WWW, or back-end databases.

Security issues mainly prevent unlimited access to a wide number of heterogeneous databases. We believe that this issue needs only to be addressed within an environment in which databases can be connected and used without violation of their security policies. This (federated) environment ensures that all its component databases can be used separately or in combination without compromising global and local security policies.

The Distributed Object Kernel (DOK) project [15] at Royal Melbourne Institute of Technology is concerned with the research and development of a secure database middleware to effectively search, update and combine information within a distributed and federated environment. DOK uses CORBA (Common Object Request Broker) technology, the distributed-object standard developed by the OMG (Object Management Group), to communicate across different database platforms. In addition, DOK provides federated services allowing clients to use multiple databases in combination, and these involve *query service* [11], *reengineering service* [14], and *reflection service* [2].

This paper addresses the design of the DOK *security service*, including the development of a secure architecture to effectively enforce federated security policies in the context of autonomous, distributed and heterogeneous databases. Users can access, update or combine data from different databases involved in a federation through a DOK layer. The security service is responsible for a detection of any violation of local and federated security policies, and triggering appropriate actions if such a violation is found prior to the manipulation of data. To implement such a detection mechanism, two main issues need to be addressed, and these include:

- The enforcement of local security policies: This involves the design of a federated access control

mechanism aiming for integration of different access controls which could have been imposed on local databases. Such an access control will specify and define the appropriate rights that a user is granted to access and/or update data in a distributed and heterogeneous environment.

- The enforcement of federated security policies: This concerns the design of an inference mechanism which manages security policies. It also prevents a user from obtaining a collection of data from different databases that may enable the user to infer sensitive information [7].

Regarding the first issue, we proposed in [13] a federated access control (called GAC) allowing a "generic" expression of the security requirements for databases involved in a federation. Each object defined within the DOK federation, also called a *virtual object*, has a set of associated access lists (ACLs) which specify the different access rights of its attributes. The DOK approach describes such security labels in terms of basic transactions (e.g., read/write/update) so that they become independent of any specific access control (e.g., DAC or MAC).

This paper describes a solution to the second issue identified above. This solution involves three aspects: (1) the design of a (simple) language to express different types of aggregation constraints, (2) a technique to monitor these constraints, and finally (3) a logical architecture and procedures to enforce the security policies. Our solution can be summarised as follows:

- (1) A logic-based language, called FELL (acronym of FEderated Logic Language), is designed to model different types of violation of federated security policies when federated transactions are submitted. This language can describe both combination of transactions issued on a single state of a federation (called *static constraints*) as well as those issued on multiple states (called *dynamic constraints*).
- (2) When aggregation constraints are expressed on a given federation, then these are transformed into appropriate data structures, called *state transition graphs*, so their monitoring becomes efficient. The nodes of such graphs model the sub-computations of the constraints. The node labels are atomic formulas, which are used during the monitoring process, and are based on the past and/or present state(s) of a federation. Due to the limited size of the paper, the focus will be on the graphs which have at least one "true" terminal node. These graphs are called *true graphs*

and an appropriate monitoring technique is proposed for such graphs based on a "linear-way" of marking nodes. The remaining type of graphs, called *false graphs*, require a different monitoring technique in which nodes are not marked in a sequential order. [12] proposes a specific technique enabling the construction and marking of false graphs.

- (3) The DOK architecture is a three-layer architecture involving a *Coordination layer*, *Task layer* and *Database layer*. Each of these layers contains specialised agents that enforce a certain part of the federated security policies. Coordination tasks (e.g., finding an appropriate agent to process a certain request) are performed by agents such as the DOK Manager. The enforcement of the security tasks (e.g., constraint maintenance) is performed by specialised agents such as the Constraint Manager. Finally, the database functions (e.g., retrieval of information about a specific user) are implemented by the user and data agents.

This paper is organised as follows. The next section overviews the DOK environment. In section 3 we briefly describe the FELL language. A framework for monitoring constraints is proposed in section 4. The description of the appropriate security agents for the enforcement of federated security policies is given in section 5. Finally, in section 6 we conclude with the current and future work.

2 Expressing Aggregation Constraints

This section describes the syntax of the language to be used for the expression of federated security policies in a DOK environment. Prior to that, we will first introduce the different elements of a DOK application, i.e. the reference model.

2.1 The Reference Model

DOK [15] is a set of managers which oversee the smooth running of a federated system and is responsible for ensuring the operational requirements of a federation. Users interact with a federation through the local external schema of one of the component databases, implemented in the local wrapper. Users' requests involving remote data are analysed by the local wrapper and re-directed to the DOK Manager, which has to ensure proper transaction management, concurrency control and query management.

A DOK schema is defined by a set of virtual objects and relationships (such as aggregation and inheritance relationships). Contrarily to conventional objects, which define physical stored entities, virtual objects describe conceptual entities defined as aggregation of objects of a distributed system. Figure 1 illustrates an example of virtual objects in which their corresponding attributes are defined by "picking up" information from three databases, namely a *personal database* (pDB - which stores information about staff members of different department of a given university), a *student database* (stDB - which stores information about students and their results), and a *bitmap database* (bitDB - which stores pictures of both staff and students of different departments). The virtual object *Department* is built by references to information located in the databases pDB and bitDB. In a similar way, the virtual object *Student* contains three types of information: *Looks-like* (which refers to a picture in bitDB), *Personal-information* (which refers to a relation or view of stDB) and *Results* (which is a SQL query on stDB constructing the results of a student). The reader may refer to [15] for more details about the DOK reference model.

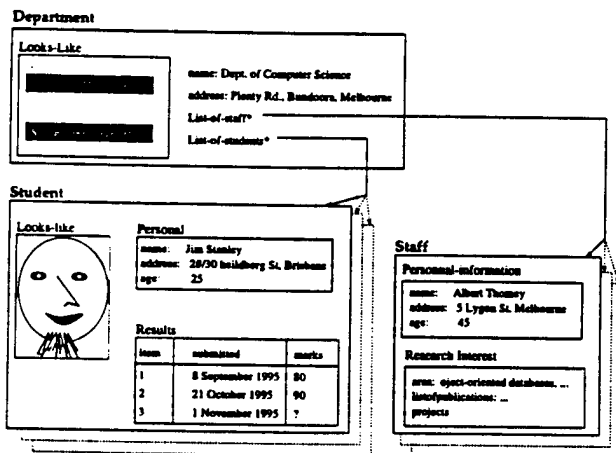


Figure 1: An Example of Virtual Objects

Using the GAC model proposed in [13], local security requirements are transformed into the federated level to allow the understanding of the local security requirements by the DOK managers. GAC is a generic access control which integrates most of the existing access controls, such as MAC and DAC.

2.2 The FELL Language

This section describes the syntax of FELL, the acronym for Federated Logic Language, which is designed to be used by a security administrator to express the different security requirements of a federation. This language provides *logic constructs* to express aggregations constraints over a database federation, however it does not provide any interpretation (or monitoring) of these constraints. As stated in the first section, the processing of the aggregation constraints is performed by generating the appropriate data structures, i.e. state transition graphs, and marking them later to detect any violation of security policies. Due to the size limitations of the paper, the semantics of the FELL expressions will be not discussed.

FELL's expressions enable the modelling of situations where a user's set of transactions are not permitted to be processed in combination on the different states of a federation, even though the user has all the appropriate access rights in the individual databases of the federation. These situations generally lead to the violation of federated security policies because the user can infer more sensitive information in which he/she does not have the required access rights [7]. The different situations which lead to the violation of federated policies are expressed as FELL's expressions, called *aggregation constraints*. In the DOK environment, we distinguish between two types of aggregation constraints: *static constraints* and *dynamic constraints*. The former are modelled as predicates on a single state of a federation, meaning that when these predicates are evaluated as true for a given state, appropriate actions are triggered. Dynamic constraints, on the other hand, are predicates specified over a sequence of states.

FELL's expressions related to static constraints are first order predicate logic. Dynamic constraints are temporal logic expressions which incorporate the temporal operators *always*, *sometimes* and *next*. In what follows, we propose some examples of static and dynamic constraints on the virtual object of Figure 1.

EXAMPLE 1 "the user u_1 cannot read the attributes salary and name of staff employee at the same time"

If we take into account different scenarios about the states of a federation, the above constraint may have two possible interpretations. The first interpretation relates to a situation where the constraint is checked within a single state (i.e., the system checks whether the user u_1 has issued a transaction in which the attribute salary and name are read at the same time).

These constraints are *static constraints*. Also, the user u_1 may have issued two transactions in which he/she reads only one attribute at the same time. This means that the constraint needs to be checked over different states of a federation. This last interpretation refers to what we call *dynamic constraints*.

2.2.1 Static Constraints

These are constraints which apply over only a single state of a federation. They express aggregations which forbid users to combine data within a single transaction. For example, the constraint stated in Example 1 is a static constraint and can be expressed as two FELL formulas:

- (1) $\forall c: \text{Staff}, \text{if read}(u_1, c.\text{name})$
implies not($\text{read}(u_1, c.\text{salary})$)
- (2) $\forall c: \text{Staff}, \text{if read}(u_1, c.\text{salary})$
implies not($\text{read}(u_1, c.\text{name})$)

These formulas basically express the order in which the user issues transactions. In (1), the user has first read the name of a staff, whereas in (2) the user has started to read the salary of the employee. In general cases, FELL provides logic constructors, such as *term*, *atomic formula* and *well formed formula* to construct logical expressions of aggregation constraints. A *term* designates either a constant, a predicate over a virtual object, a variable, or projected variable. FELL also supports the definition of *atomic formulas* as a composition of terms using specific rules. For instance, if P , Q and R are terms, then $\text{read}(P, Q)$, $\text{write}(P, Q, R)$, $\text{delete}(P)$ are atomic formulas. Using the concept of atomic formula, one can construct more complex structures of the universe of discourse. These FELL structures are called *well formed formulas* (in short, formula) and they can be built using appropriate rules.

2.2.2 Dynamic Constraints

These are defined over several states of a federation (e.g., past, present and future). They express long-term data dependencies between remote states in the evolution of a federation. Thus, the different states in a sequence must be inspected as a whole in order to detect a violation of a constraint.

Let us consider an example of a dynamic constraint. Using the example of Figure 1, one can stipulate that the only user who can update the salary of an employee is the boss of the department. This constraint can be expressed as follows:

$$\forall e, d, u, e:\text{Staff}, d:\text{Department}, u:\text{User},$$

$$\text{always } (e \in d.\text{staff} \text{ and } d.\text{boss} = u)$$

$$\text{before write}(u, e.\text{salary}, -)$$

Another alternative is to allow the update of the salary of an employee only by people who have been the head of a department at least once. This constraint relaxes the previous formula in which only one state of a federation is required to satisfy the assumption related to the fact that a user has been a boss of a department. This constraint is expressed as follows:

$$\forall e, d, u, e:\text{Staff}, d:\text{Department}, u:\text{User},$$

$$\text{sometimes } (e \in d.\text{staff} \text{ and } d.\text{boss} = u)$$

$$\text{before write}(u, e.\text{salary}, -)$$

3 State Transition Graphs

So far we have presented some of the concepts related to the DOK model and the FELL language. This section deals with the analysis of aggregation constraints.

Analysis of constraints consists of both detecting inconsistencies and transforming them into a form that can be monitored. The task of the monitor is to check whether federation states are admissible in the context of a certain federation history. This task is performed using *state transition graphs* to identify any violation of security policies. These graphs consist of:

- *Labelled nodes* with temporal formulas. The initial node is labelled with the constraint itself. The nodes reflect the conditions which have to be fulfilled by the future states of database objects.
- *Labelled edges* with non-temporal ones. The edges reflect the transition between constraints, and their labels indicate under which conditions such a transition may occur.

Before going into a detailed description of the consistency checking mechanism, we first give a few examples of state transition graphs associated with aggregation constraints. Four types of constraints are described.

CASE 1: The constraint in the form of [*sometimes* ψ]. The following is an example of such constraint

$$\forall s, \exists u, s:\text{Student}, u:\text{User},$$

$$\text{sometimes}(\text{write}(u, s.\text{Results}, -))$$

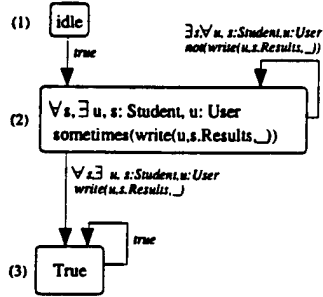


Figure 2: “sometimes” predicate

which stipulates that there exists a user who can update the results of all students. The type *User* is a predefined type and used for defining FELL expressions. The corresponding state transition graph is shown in Figure 2.

Each student, say *e*, belongs to one of the two nodes of the graph illustrated in Figure 2. We assume that the constraint is in the form of [*sometimes* ψ].

- If in the current state of a federation, the object *e* does not verify the formula ψ , then *e* will be used again in the future states to check ψ . In this situation, the validation process of the constraint is still in the node (2) and the constraint can be re-expressed as [*existnext* ψ], meaning that ψ is required to be checked in all future states.
- If *e* verifies the formula ψ , then the corresponding temporal constraint is valid. The node (3) of the state transition graph is reached.

CASE 2: This case deals with temporal formulas in the form of [*always* ψ]. The state transition graph of this type of constraint is different from the previous one because the temporal formula ψ needs to be valid in every future state of a federation. Figure 3 illustrates the graph of the following constraint.

$\forall s, \exists u, s: \text{Student}, u: \text{User},$
 $\text{always}(\text{write}(u.s.\text{Results},_))$

In a given state of a federation

- if any object *s* of type Student can be updated by a given user, say *u*, then the constraint of type [*always* ψ] becomes true in the current state of a federation. Additionally, the constraint ψ is required to be valid in all the future states. This constraint is re-expressed as [*allnext* ψ].

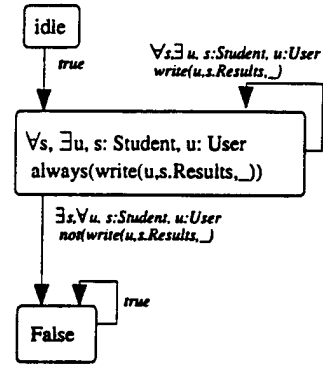


Figure 3: “always” predicate

- if the formula ψ is not fulfilled in the current state, i.e there exists an object *s* which cannot be updated by any user, then the constraint *always* ψ is no longer valid.

CASE 3: This case is concerned with the order of validation of constraints. One situation can be in the form of [*always* ψ before ϕ], where ψ and ϕ are non-temporal constraints. Let us consider the following constraint:

$\forall s, \forall u, s: \text{Student}, u: \text{User},$
 $\text{always}(\text{read}(u.s.\text{name})) \text{ before}$
 $\text{write}(u.s.\text{Results},_)$

This constraint specifies that a modification of a student's results is authorised only if the user has already checked the name of the student. The state transition graph of such a constraint is shown in Figure 4.

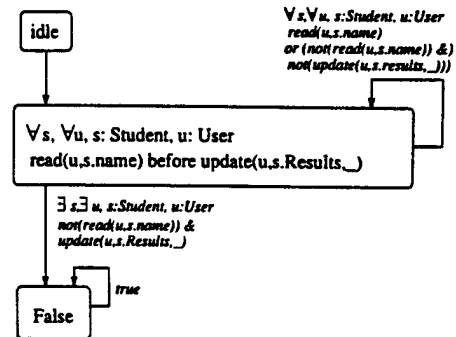


Figure 4: “always – before” predicate

CASE 4: The constraint is in a form of [sometimes ψ before ϕ]. The following example illustrates such types of constraints.

$\forall s, \forall u, s:\text{Student}, u:\text{User},$
 sometimes(read($u, s.\text{name}$)) before
 write($u, s.\text{Results}, _$)

Figure 5 describes the state transition graph for the above constraint.

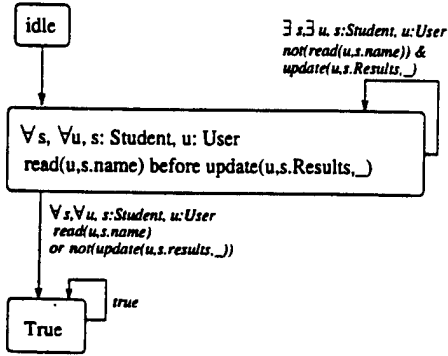


Figure 5: “sometimes – before” predicate

The basic state transition graphs for temporal formulas are shown in Figure 6. Using these graphs, one can build state transition graphs for complex constraints over multiple states of a federation. These complex constraints are built from other constraints by logical composition of atomic formulas. Thus, the state transition graphs of such constraints are built, in a similar way to their logic representations, by aggregation of the basic state transition graphs shown on Figure 6.

However, a distinction between aggregation of state transition graphs having no *True* terminal nodes and those that do have, needs to be made. This distinction between these types of graph is important because of the differences in the way their corresponding formulas are to be monitored. The first category of graphs relates to formulas that could not be always true after a certain state of a federation. This is mainly because these graphs do not contain a *True* terminal node. We call these graphs **false graphs**. The second category of graphs involves those which contain a *True* terminal node. The *True* node of these graphs can be reached in some given state of a federation, meaning that the corresponding formulas are and will be validated in all the future states of a federation. These graphs are called **true graphs**.

False graphs have been studied in [12], where algorithms for building and monitoring such graphs are

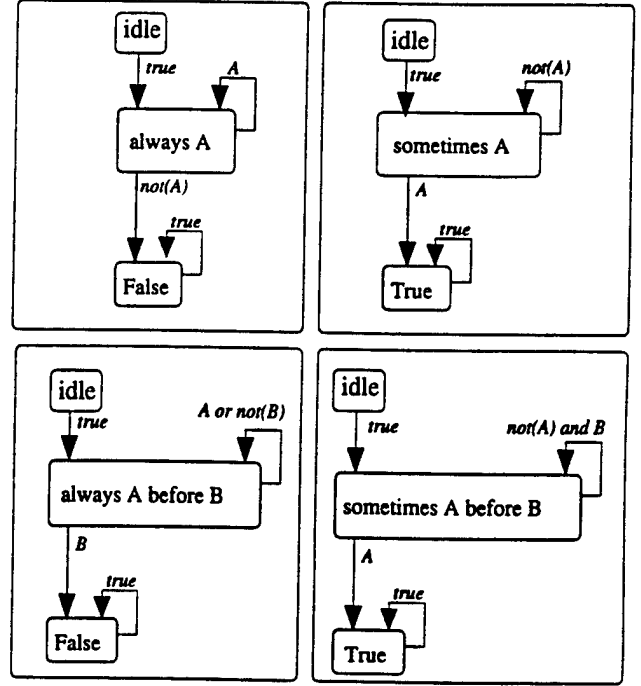


Figure 6: The Basic State Transition Graphs

provided. This paper deals with true graphs.

3.1 True Graph Composition

Let us first consider an example of constraints which are based on true graphs. The following constraint stipulates that the user u can never aggregate the attributes *salary* of an employee and the *name* of the department where the employee is working.

$\forall e, \forall d, e:\text{Staff}, d:\text{Department}$
 sometimes(read($u, d.\text{name}$)) \wedge
 sometimes(read($u, e.\text{salary}$)) \wedge
 e is-in $d.\text{staff}$ implies abort

The above formula is a logical composition of three formulas: (ψ_1) [sometimes(read($u, d.\text{name}$))], (ψ_2) [sometimes(read($u, e.\text{salary}$))] and (ψ_3) [(e is-in $d.\text{staff}$)]. Using the basic state transition graphs (shown in Figure 6), the state transition graphs of such atomic formulas can be built. Figure 7 illustrates these graphs, denoted as G_1 , G_2 and G_3 . Note the graph G_3 is a simple graph because the corresponding formula checks in each state of a federation whether the user u is reading information about a staff member of a department. The state transition graph for the complex constraint $\psi_1 \wedge \psi_2 \wedge \psi_3$ is defined as a set of all possible aggregations of the graphs G_1 , G_2 and G_3 , i.e. $\{G_1 \oplus G_2$

$\oplus G_3, G_1 \oplus G_3 \oplus G_2, G_2 \oplus G_1 \oplus G_3, G_2 \oplus G_3 \oplus G_1, G_3 \oplus G_1 \oplus G_2, G_3 \oplus G_2 \oplus G_1$, where \oplus represents the graph aggregation operator. This operator connects the nodes of the concerned graphs to form a complex state transition graph. Formally, this is defined as follows:

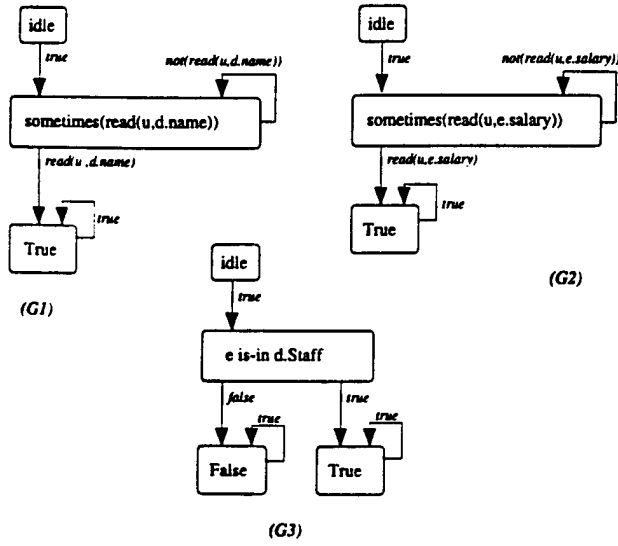


Figure 7: Examples of State Transition Graphs

ALGORITHM 1 (Graph Composition) Let us consider two true graphs $G_i = \langle \{n_{i_1}, \dots, n_{i_p}\}, \{t_{i_1}, \dots, t_{i_{p-1}}\} \rangle$ and $G_j = \langle \{n_{j_1}, \dots, n_{j_k}\}, \{t_{j_1}, \dots, t_{j_{k-1}}\} \rangle$, where n_x is a node type and t_y is a transition. We assume that these graphs are state transition graphs of the formulas ψ_i respectively ψ_j . The composition of G_i and G_j , denoted as $G_i \oplus G_j$, produces the state transition graph $G_{ij} = \langle \mathcal{I}, \mathcal{J} \rangle$ for the formula $\psi_{ij} = \psi_i \wedge \psi_j$, where

1. $\mathcal{I} = \{n_{i_1}, \dots, n_{i_p}, n_{j_1}, \dots, n_{j_k}\}$ is the set of all nodes of the two graphs G_i and G_j .
2. $\mathcal{J} = \{t_{i_1}, \dots, t_{i_{p-1}}, t_{new}, t_{j_1}, \dots, t_{j_{k-1}}\}$ is the set of transitions of G_{ij} . The transition t_{new} from n_{i_k} to n_{j_r} of G_i and G_j respectively is defined as follows:

- (a) find a node of G_i which is labelled as True, say n_{i_k} ;
- (b) delete the cyclic transition of the node n_{i_k} ;
- (c) if n_{j_r} is the first node that is reached from the idle node in G_j , then create a transition t_{new} from n_{i_k} to n_{j_r} ; and

(d) label the transition t_{new} as true.

A True node of the graph G_i is used as a basis to build the complex graph G_{ij} . During the checking of the validity of the complex formula ψ_{ij} , if the True node of G_i has already been reached (or marked as we will see later), then the next step in the processing of ψ_{ij} will be to check the formula ψ_j using the graph G_j . Since the True terminal node of the graph G_i has a cyclic transition, and to allow a transition from G_i to G_j , this cyclic transition needs to be deleted (as stated in 2-b).

The last stage in building the graph G_{ij} is to make the transition between the True node of G_i and one of the nodes of the graph G_j . As stated in the condition 2-c, the first node in G_j which is reached after the idle node is used in the next stage of the processing of the formula ψ_{ij} . Finally, the conditions (2-c and 2-d) create and label the transition t_{new} with true to allow a direct transition from G_i and G_j .

EXAMPLE 2 Figure 8 shows some of the composition performed on the graphs G_1, G_2 and G_3 of Figure 7.

The graph generated by using the composition operator \oplus can have some redundant nodes that are not required to be checked during the monitoring of aggregation constraints. These nodes relate to the non-terminal True nodes that have been used during the composition to link different graphs. If we consider for instance the composition graph $G_1 \oplus G_2 \oplus G_3$ of Figure 8, we notice that some nodes of such a graph are always true. This makes their checking unnecessary. These nodes basically allow the transition from the checking of one formula (i.e., ψ_i in our case) to the checking of another formula (i.e., ψ_j). The terminal True nodes cannot be deleted because they related to the situation where the complex formula is proven to be true.

ALGORITHM 2 (Graph Simplification) Given a graph $G = \langle \mathcal{I}, \mathcal{J} \rangle$ and a non-terminal True node n_i connecting two nodes n_{i-1} and n_{i+1} with transitions t_{i-1} and t_{i+1} respectively, then we construct an equivalent graph $G^1 = \langle \mathcal{I} - \{n_i\}, \mathcal{J} - \{true\} \rangle$ of G , where

- true is the label of the transition from n_i to n_{i+1} ;
- t_{i-1} is a transition from n_{i-1} to n_{i+1}

is an equivalent to G .

EXAMPLE 3 Figure 9 shows two successive simplifications of the initial state transition graph $G = G_1 \oplus G_2 \oplus G_3$ of Figure 8, denoted as G^1 and G^2 .

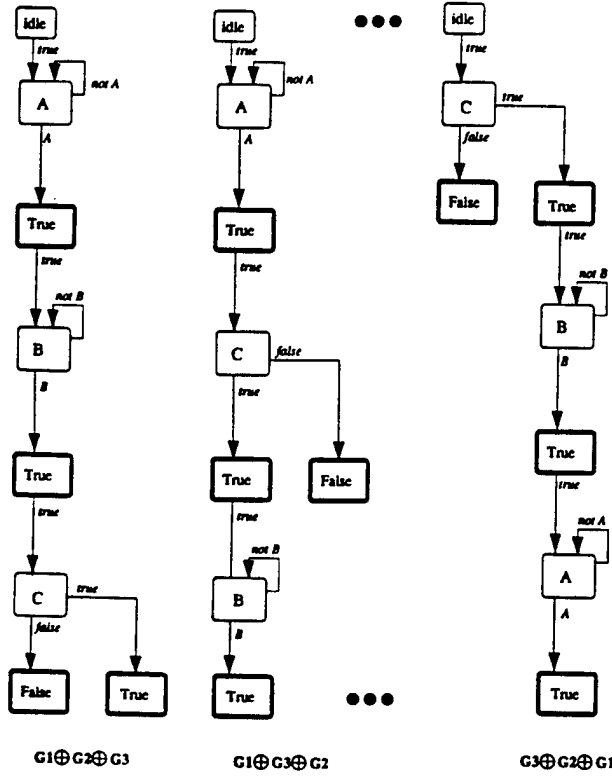


Figure 8: The composition of graphs G_1 , G_2 and G_3 , where $A = \text{sometimes}(\text{read}(u_1, d.\text{name}))$, $B = \text{sometimes}(\text{read}(u_1, e.\text{salary}))$, and $C = e \text{ is-in } d.\text{Staff}$

Note that the graph G^2 cannot be simplified anymore even though it has a *True* node. This node is a terminal node and describes the final state of the computation of the complex formula $\psi_1 \wedge \psi_2 \wedge \psi_3$. The graph G^2 is called a *reduced graph* of G .

4 Marking Algorithms

Monitoring aggregation constraints consists of checking whether the federation states are admissible in the context of a certain federation history. In this section we describe two algorithms which enable the monitoring of aggregation constraints by marking the nodes of state transition graphs.

For a given complex formula, e.g., $\psi = \psi_1 \wedge \psi_2 \wedge \psi_3$, there may exist several state transition graphs (as shown in Figure 8). Thus, many reduced graphs can be derived. The graph G^2 is an example of a reduced graph of the constraint ψ . The checking of a complex formula is performed on the reduced state transition graphs generated from its initial state transition

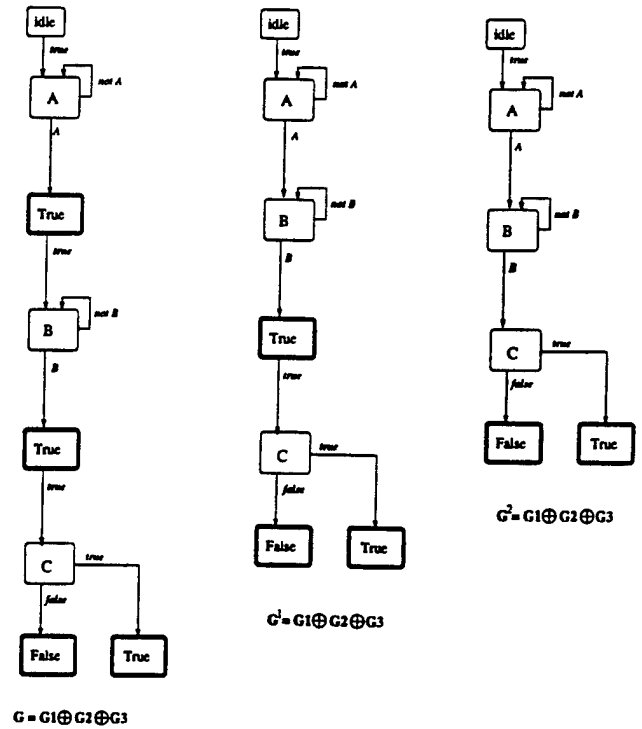


Figure 9: The Simplification of $G = G_1 \oplus G_2 \oplus G_3$, where $A = \text{sometimes}(\text{read}(u_1, d.\text{name}))$, $B = \text{sometimes}(\text{read}(u_1, e.\text{salary}))$, and $C = e \text{ is-in } d.\text{Staff}$

graphs by marking the nodes that have been found true in all past states of a federation. Using the example of Figure 9, the checking of the constraint ψ can be done by marking the nodes of the different reduced graphs. For instance, if we assume that the node A has been validated in one of the past states of a federation, i.e. the user u has already read the name of a department (say d), then there are two possible scenarios to mark the remaining nodes of the state transition graphs:

1. either the user u is reading the salary of an employee, say e , in the current state of a federation. This means that the node B will be marked, and the next step in the checking of the constraint ψ is to evaluate the node labelled as C ;
2. or the node labelled by C needs to be evaluated before checking the node labelled by B ; i.e. check whether the staff member (in which the user u has read its information) is working in a department (which has been read by the user u).

The first scenario follows a *linear* marking of the reduced graphs. We call such a marking technique LMT

(*Linear Marking Technique*), and a node is marked if and only if all its precedent nodes have been marked either in the past or current states of a federation. In contrast to the first scenario, the second scenario follows an *anarchic* way of marking (called *Zigzag Marking Technique*), where any node of a reduced graph can be marked when its corresponding formula is made true in the current state.

4.1 Linear Marking Technique

Let us consider a formula ψ with $G^1, \dots, G^i, \dots, G^n$ as a set of all its reduced (true) graphs. We assume that these graphs are defined as follows: $G^1 = \langle \{n_{1_1}, \dots, n_{1_k}\}, \{t_{1_1}, \dots, t_{1_{k-1}}\} \rangle, \dots, G^i = \langle \{n_{i_1}, \dots, n_{i_k}\}, \{t_{i_1}, \dots, t_{i_{k-1}}\} \rangle, \dots, G^n = \langle \{n_{n_1}, \dots, n_{n_k}\}, \{t_{n_1}, \dots, t_{n_{k-1}}\} \rangle$. We also assume that the nodes of these graphs can be marked by a function, namely **mark**: $\mathcal{N} \rightarrow \{+, ?\}$, where \mathcal{N} is the set of possible nodes. In the initialisation phase, the nodes of the reduced graphs are marked as follows:

```
% initialisation phase %
for each i,  $1 \leq i \leq m$ 
    for each j,  $1 \leq j \leq k$ 
        if label( $n_{ij}$ )  $\neq$  True
            then mark( $n_{ij}$ ) = ?
            else mark( $n_{ij}$ ) = +
```

where the function **label** returns the label of a node. In the above initialisation phase, all the nodes of the reduced true graphs (except the nodes labelled with *True*) are marked with the symbol "?", meaning that their corresponding formulas have not been validated. The nodes having *True* as a label are marked with the symbol "+".

We assume that at time t , the reduced graphs have been marked according to the different events that occurred in the past states of a federation. At time $t + \Delta t$, which is the current state of the federation, an event occurs where the user is reading, updating, or combining information from different virtual objects. We denote these events by the different sub-transactions (i.e. *read* and *write*) which are initiated by the user of the federation. We denote these sub-transactions as s_1, \dots, s_h . The LMT algorithm checks the different reduced graphs and traverses the nodes marked with the symbol "?". The nodes that match with the sub-transactions s_1, \dots, s_h are marked with the symbol "+", otherwise no modification in the marking of the nodes is performed.

% LMT Algorithm for True Graphs %

Input: The following input data is assumed

- a complex formula ψ
- ψ 's reduced graphs $G^1 = \langle \{n_{1_1}, \dots, n_{1_k}\} \rangle, \dots, G^m = \langle \{n_{m_1}, \dots, n_{m_k}\} \rangle$
- a set of sub-transaction s_1, \dots, s_h
- variable *status* which has a value 1 when ψ becomes true in the current state of a federation, 0 otherwise.

Output: value of *status*

Procedure: The following steps are performed
for each $i = 1, m$ do

- (1) find the last node of G^i marked with +, n_x^i
- (2) find the next node of n_x^i in G^i , say n_y^i
- (3) evaluate the label of n_y^i , say $\psi_{n_y^i}$, against s_1, \dots, s_h

if $\psi_{n_y^i}$ is true
then

- (4) mark(n_y^i) = +
- (5) find the next node of n_y^i in G^i , say n_z^i
- (6) if label(n_z^i) = true
then
 - (7-1) status = 1
 - (7-2) exit
- (8) else status = 0

The above LMT algorithm marks the nodes of a reduced true graph in a linear manner. For a given reduced graph, the step (1) finds the last node that has been marked with the symbol "+", denoted as n_x^i . The step (2) evaluates the label of this node against the different sub-transactions that have been issued in the current state of a federation. This evaluation step checks whether or not the sub-transactions are part of the label of the node n_y^i , and then evaluates their logical combination (depending on the complex formula $\psi_{n_y^i}$ in the node n_y^i). If the formula becomes true, then the next step consists in checking whether the *True* node has been reached, meaning that there exists at least one reduced graph which has got all its nodes marked with the symbol "+". The steps (7-1) and (7-2) are important steps of the LMT algorithm because they avoid further marking of the remaining reduced graphs when at least one of the reduced graph has all its nodes marked with the symbol "+".

4.2 The Zigzag Marking Technique

Now we shall introduce the remaining algorithm for the true graphs. This algorithm is based on a non-linear search of a node that has not been marked with the symbol "+". In contrast to the linear search of the LMT algorithm, the ZMT algorithm retrieves any node of a reduced graph

when the label of this node can be matched with the set of sub-transactions issued in the current state of the federation. When all the nodes of a given reduced graph are marked with the symbol "+", the algorithm exits in the similar way as LMT because the formula has been validated within the current state of the federation.

% ZMT Algorithm for True Graphs %

Input: The following information are assumed

- a complex formula ϕ
- reduced graphs $G^1 = \langle \{n_{i_1}, \dots, n_{i_k}\} \rangle, \dots, G^m = \langle \{n_{m_1}, \dots, n_{m_k}\} \rangle$ of ϕ
- issued sub-transactions s_1, \dots, s_h
- variable *status*

Output: value of *status*

Procedure: The following steps are performed
for each $i = 1, n$ do

for each $j = 1, k$ do

if $\text{mark}(n_{i_j}) = ?$

(1) evaluate the label of n_{i_j} , say $\phi_{n_{i_j}}$, against s_1, \dots, s_h

(2) if $\phi_{n_{i_j}}$ is true

(3) then $\text{mark}(n_{i_j}) = +$

(4) check all the labels of n_{i_1}, \dots, n_{i_k}

(5) if all labels of the nodes is True

(6) then exit

Contrary to the LMT, the above algorithm checks all the nodes of a reduced true graph which have "?" as labels. If some of these nodes are evaluated to true against the current sub-transactions, then these are marked with the label "True". Thus, the algorithm does mark (as opposed to the LMT) several nodes at the same time during the processing of a formula. This can be an advantage over the LMT algorithm, however the complexity of the ZMT increases when the state transition graphs become large (i.e., n becomes very important). Finally, the complexity of the above algorithm is $O(n! \times c_t)$, where c_t represents the cost of matching a set of sub-transactions against a set of atomic formulas. The ZMT algorithm is useful for false graphs but not for true graphs (unless the number of nodes is very small) because every node of a true state transition graph must be marked [12].

5 Security Agents

This section addresses the design of a secure DOK architecture to support the framework presented in the previous sessions. As mentioned earlier in this paper, DOK [15] is a system providing federated services, such as a *reengineering service* [14], *query service* [11], *reflection service* [2], *trader service*, *security service*, and *transaction service*. Each of these federated services is implemented as a server

and it is used by different processes to perform specific functions, such as retrieving objects, processing queries over a set of databases, etc.

To implement the different functions of each of the DOK services, we have designed a logical and physical architectures aiming to support interoperability across different database platforms. The logical architecture describes the DOK layers and the intra- and interaction between these layers to allow efficient communication and processing of the different functions of the DOK services. The physical architecture describes the implementation of the different components of the logical architecture. This section focuses on the DOK logical architecture.

The DOK logical architecture is based on the use of *agents* [4] to perform the functions of the DOK services. Also, these agents are designed to be able to understand and abstract (through a reflective process) information embedded within different applications of a federation, negotiate with remote agents to perform in collaboration different activities of a federation, etc. In this section we will describe the DOK security agents and relate their activities to the algorithms proposed in the previous sections.

5.1 The DOK Agent Model

A DOK *agent* refers to an active entity which performs specific tasks within a federation such as enforcing security, ensuring the committing of global transactions, mining resources or optimising global queries. In the DOK agent model, an agent is graphically represented with (circle/rectangle/etc) icon (see Figure 10) and it contains different information, such as the *name* of the agent, a set of properties (which for example represents the information required for enforcing security policies) and the corresponding methods or functions (that is the procedures allowing to keep a federation secure).

The interaction between agents is based on the different relationships they have between them. The stronger the relationship between two agents, the bigger is the communication between them. Our model supports three types of (communication) relationships: *containment relationship*, *association relationship*, and *inheritance relationship*. These relationships are supported by most of object-oriented models [10], however our relationships are related to dynamic issues (e.g. as communication between agents) than static issues (e.g., attributes factorisation). More importantly, these introduced relationships are close to those that exist between humans. In this way, containment relationships are parental relationships allowing the expression of a relationship between a father/mother and their children. The association relationships is more-or-less like neighbourhood relationships. They express some sort of relationship between human agents, however they are weaker than containment relationships. Inheritance relationships express a kind of specialisation of human agents to perform specific tasks. An example of such a relationship is for instance the relationship between a staff member and a teacher (or an administrator). A staff member is a

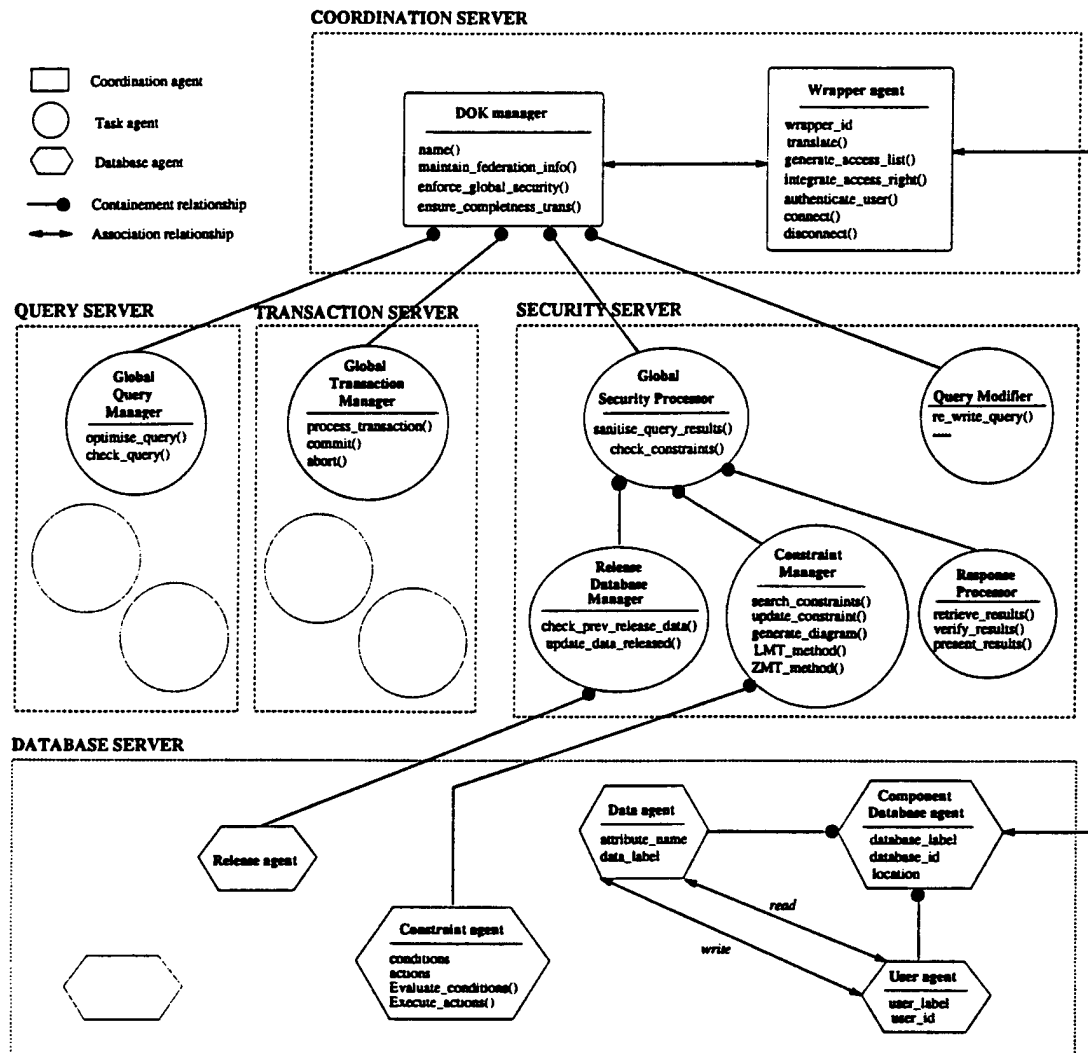


Figure 10: The DOK Logical Architecture

generic agent who can perform some functions, however a teacher is specialised in providing lectures for students in the same way that an administrator is specialised in performing administrative work.

Even though the relationships between agents look like those of object-oriented models, they are different in underlying semantics. A good example which justifies this claim is the following. In our approach, if we want to express a relationship between, for example, an agent which represents a president of a country and its minister agents, we will use the containment relationships because they express the sub-computations that the minister agents are responsible to perform on behalf of the president agent. However, in object-oriented data models, these relationships will be represented as association relationships. As

we will see later, we will use an object-oriented model only to represent the data used by agents during the communication process (by sending for example an object identity of an object located in a given site). In this way, the agent model describes the agent information and behaviour, whereas object-oriented models are used to describe the data as well as operations which can be applied or used by agents.

The communication between a father agent and its children can be qualified as a "strong relationship". The "weaker relationships" are modelled as associations. They also express inter-dependencies between agents, however they are related to a mutual collaboration between agents in order to perform one or multiple common functions. Referring again to Figure 10, the association relationship

between the *DOK Manager* and the *Wrapper agent* represents a sort of collaboration in which one of these agents can request, for example, to transform a schema of a given database in order to be understood by the different agents of a federation. Agents manipulate data which are in the form of objects [10]. In this way, a schema of a relational database will be transformed into an object-oriented schema by using one of the functions of the wrapper proposed in [14]. Note that the Wrapper agent may communicate with the DOK Manager and request, for instance, some of its services.

The last relationship is the *inheritance relationship* between agents. The schema of Figure 10 does not have one, however a simple example of such a relationship can be defined between the Wrapper agent of Figure 10 and another new agent which will model the Wrapper for a specific database, e.g. an Oracle database. We call this new agent, an *WrapperOracle* agent, and this will be like any other wrapper, that it can perform any of the functions of an ordinary wrapper. However the differences with the other wrapper agents is that *WrapperOracle* is specialised for Oracle databases, that is it can transform (or translate) only schema defined with an Oracle database. In the same way, this wrapper only has an understanding of the security information (e.g. access control) of Oracle databases.

5.2 The Agents

As for other functions of the DOK system, the enforcement of the security policies by the DOK agents is a multi-level process. At the *top level*, the DOK Manager or Wrapper agents (depending on the level of the security checking) identifies the type of task to perform in the federation (e.g. authentication). The agents of this top level are aware about all the activities which are happening (or already happened) in the federation. Also they can access information about each agent (e.g., roles, functions) and request other agents to process some functions. In this way, the top layer of the DOK involves what we call *coordination agents*. They are responsible for the coordination of all activities of a federation rather than performing the tasks themselves. These tasks (or functions) are instead delegated to more specialised agents of lower levels. At the *middle level*, specific functions (such as enforcement of global constraints or the sanitisation of query results) are performed by what we call *security agents*. Examples of such agents are the Global Security Processor or Constraint Manager of Figure 10.

Security agents differ from coordination agents because they are specialised in performing specific functions of the DOK system. Thus the security agents have a narrow visibility of a federation, that is they have knowledge only about agents which perform the same function. The *bottom level* is comprised of a set of agents specialised in accessing or updating information required by agents of higher layers. These agents are database-like agents playing the role of an interface between the participating databases and the agents of top and middle layers. They

are called *database agents*. An example of such agents is the User agent which records all information about a particular user, including the different access rights that he/she has for different objects as well as the identity of the user.

This three-layered architecture of the DOK system has many advantages. Each layer involves a set of agents which are responsible for a certain activities in a federation. These activities can be management activities, that is they are performed by coordination agents which will oversee the running of a federation. Other activities will be, for example, those which relate to a specific task. These activities are performed by specialised agents. In this way, specialised agents have a narrow view of the whole system. Finally, the last activities concern more simpler functions which are related to the "preparation" for instance data to be used either by specialised or coordination agents, the storage of data related to objects of local databases, etc. This classification of agents based on their activities in a federation shows a sort of a logical clustering (or modules) of the DOK agents. Figure 10 illustrates the different DOK clusters, that is the *coordination server*, *security server*, *transaction server*, *query server*, *database server*, etc.

As mentioned earlier, the agents of the different DOK modules interact to perform the required functions of the DOK system. The collaboration between the agents of the same cluster (server) is larger and more intense than those which belong to different clusters. The main reason for this is that if two agents are involved in performing the same function, they should be related by stronger relationships, i.e. the containment relationship. In the opposite case, the two agents will be involved in a "weaker" collaboration. This means that they need to be defined in different modules. Since this article is only related to security enforcement, we have illustrated in Figure 11 the interaction only between security agents in order to enforce both local and federated security policies.

5.2.1 Coordination Layer

This level involves a set of agents specialised in the coordination of the different activities of a federation. With regard to security enforcement, the DOK Manager and the Wrapper agent are the only agents with the overall understanding on how to keep a federation secure. When a query is sent to a local database, the wrapper authenticates the user and determines whether the query is related to the local database or to the federation. If it is a local query, the wrapper will delegate the processing to the local database. Otherwise, the query is translated from a language used at the local level into a global level [14] and then processed by the the DOK Manager.

WRAPPER AGENT

The role of this agent is to translate the query information from the local level to the global level and vice versa. This

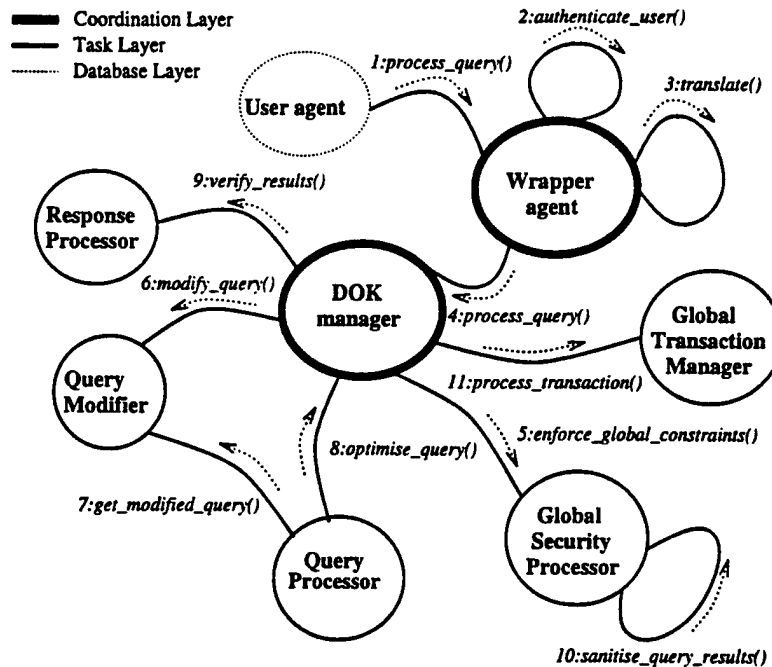


Figure 11: Collaboration between DOK Agents

will allow the DOK Manager and other agents to have a better understanding of information contained within local databases. In terms of security, the Wrapper is responsible for authenticating local users at the global level before presenting the query. The wrapper agent checks the security attributes of the user, and relays them to the DOK Manager for enforcement of global security. This is depicted by the Component Database, User and Data agents, and the relationships among these agents (see Figure 10). The methods provided by the Wrapper agent are *authenticate_user()*, *connect()*, *disconnect()* and *translate()*. The method *authenticate_user()* invokes communication with the User, Data and Component Database agents to gather information about the types of access rights a user has to a certain data when interrogating other component databases.

The methods *connect()* and *disconnect()* establish the communication between the wrapper and the federated level by connecting or disconnecting a database from a federation, whereas the method *translate()* will do the query translation from the local to the global level using the algorithms provided in [14]. The methods *generate_access_list()* and *integrate_access_right()* allow the mapping and the integration of security information of local databases into the federated level. These methods have been described in the previous section.

DOK MANAGER

This agent allows the creation of DOK instances for specific federated environments. The DOK Manager contains an attribute name (e.g.: *banking* or *schooling*) for identification purposes. Every instance of this agent runs on a specific federation and can communicate with other federations. Most of the functions of the DOK Manager agent are performed by delegating to other agents. For instance, a DOK Manager ensures the completeness and correctness of global transactions by requesting the execution of methods of the *Global Transaction Manager* and the *Global Query Processor*. More importantly, since we are discussing security, the DOK Manager would solicit the methods of the *Global Security Processor* to enforce global security. Let us consider in detail the security procedures of the DOK Manager. When a request is sent by the wrapper to the DOK Manager to process a query, as shown in Figure 11, the DOK Manager performs the following steps:

- Checks whether a user can access individual aggregates specified in the query. At this level, the global access control mechanism has been used to generate the federated access list for each aggregate of a virtual object. The methods *generate_access_list()* and *integrate_access_right()* are described in [13]. The DOK Manager requests wrappers to generate the access lists for each local aggregate of the participating databases. The generated access lists are then integrated to become the federated access list of the ag-

gregate. The DOK system will grant or deny access or update of the information to the user according to the federated access list.

- When a user is effectively allowed to access or update individual aggregates specified in the query, the second step of the query processing consists of checking whether or not the user is allowed to combine aggregates to derive information denied to him/her. At this stage of the pre-processing of the query, the DOK Manager will delegate the enforcement of the constraints to the Constraint Manager which will look for all types of constraints related to the corresponding user. When a set of constraints are found, the query is re-written by the Query Modifier to include the constraints within the query.

5.2.2 Task Layer

At this level, agents perform specific tasks to ensure that all the aspects of security processing are carried out properly to maintain global security. The tasks of maintaining federated security policies are delegated by the DOK Manager to specialised agents such as the *Global Security Processor*, *Query Modifier*, *Release Database Manager*, *Release Database agent* and the *Response Processor*. Here we focus on the agents responsible for the enforcement of aggregation constraints.

GLOBAL SECURITY PROCESSOR (GSP)

The main role of this agent is to assist the DOK Manager in enforcing federated security policies. It does this via methods such as *enforce_constraints()*, *build_graph()*, *mark_graphs()*, and *sanitise_query_results()*. The method *check_constraints()* preprocesses, where possible, the security aspects of a global query before they are sent out to the affected sites for execution. This does not include constraints dependent on the result of the query, such as constraints restricting the number of instances of an object or facet that a subject is allowed to retrieve in a query. The *sanitise_query_results()* method post-processes the results of the query to enforce constraints of this type. The execution of each of the methods described above results in the communication between agents. The agents involved include the Query Modifier, Response Processor, Release Database Manager, Release Database, Constraint Manager, Constraint and Wrapper agents.

The remaining methods of the GSP agent relate to the enforcement of aggregation constraints. This is done through the triggering of the method *check_constraints()* which is delegated to the Constraint Manager.

CONSTRAINT MANAGER

When an event is received from the GSP agent, the Constraint Manager performs the following tasks:

- (i) it finds appropriate constraints for a given event in a federation using *search_constraints()*;

- (ii) it builds the state transition graphs for the selected constraints using *generate_graph()*. This method is based on Algorithms 1 & 2 of Section 3.

- (iii) it monitors the selected constraints using the methods *LMT_method()* and *ZMT_method()* of Section 4. If any violation of constraints is detected, then actions are triggered by the Constraint agent.

The sanitisation of queries is delegated by the GSP agent to the Constraint Manager. The processing of these specific constraints (i.e. those that are concerned with the sanitisation of query results) is performed in similar to aggregation constraints. The only difference between the processing of aggregation constraints and sanitisation constraints is that the former is performed before any evaluation of the user query, whereas the later is done after.

6 Concluding Remarks

In the area of distributed databases, much work has been focussed on providing appropriate access control [6, 5]. However, aggregation in distributed databases is currently the most difficult and challenging problem. This deals with the issue of inferring data classified as high level (or *high data*) from some set of data classified as a low level (or *low data*) [7]. That is, there is a direct inference path (possibly including external data) from the low data to the high data.

Existing solutions for the aggregation problem can be classified according to the type of inference channel to be detected. As discussed in [7], three types of channels can be detected: *logical inference channels*, *abductive inference channels*, and *probabilistic channels*.

1. Approaches based on the detection of logical channels focus on the construction of formal deductive proofs showing the existence of the derivation of high data from a low data. The proposed solutions, mainly elaborated by the researchers of the Computer Science Laboratory at SRI International, deal with the use of formal theorem proving for detecting inference channels [9].
2. A slightly weakened requirement for a logical channel is when a deductive proof may be not possible, but a proof could be completed by assumption of a certain axioms. The development of an abductive proof takes into account the degree to which a user is likely to know some facts necessary to the completion of a proof.

The approaches for the detection of abductive channels are based on epistemic logics [3] which "relax" the conventional methods of reasoning (theorem proving) to include the user's beliefs as a part of the model.

3. The detection of probabilistic channels is based on the inference of high data with some measure of belief greater than an acceptable limit. Buczkowski, in [1], uses a probabilistic model (based on Bayesian probability) to estimate security risk due to partial inference.

The above approaches (except the Probabilistic one) are based on constructing a proof (or building a model, that is the generation of all possible high sensitive data [8]). These approaches are formally sound and are useful in detecting different channels. However we believe that they are limited in their use for large scale applications, particularly in distributed environments.

The proposed approach has advantages and disadvantages. One of advantages of the DOK security approach is that it is based on computational issues of aggregation constraints instead of building formal proofs. Our approach builds appropriate data structures (i.e., *state transition graphs*) to model the different sub-computations of aggregation constraints. These sub-computations are represented as nodes labelled with atomic (temporal) formulas. A marking technique is proposed to monitor such constraints depending on the type of state transition graphs.

The limitations of the DOK approach regarding the enforcement of federated security policies relate to the complexity of the construction and the marking of state transition graphs. For simple aggregation constraints, the proposed approach is very efficient. However for more complex constraints, appropriate access methods are needed to access a large database of nodes (of state transition graphs) to enable efficient enforcement of federated security policies.

We are currently implementing the proposed security framework using NEO and KQML¹. Coordination agents are being first implemented and the remaining (security) agents are designed based on the algorithms proposed in previous sections.

References

- [1] L.J. Buczowski, "Database Inference Controller," In: *Database Security III: Status and Prospects*, D.L. Spooner and C. Landwehr (eds.), North-Holland, 1990.
- [2] D. Edmond, M. Pazoglou, and Z. Tari, "An Overview of Reflection and Its Use in Cooperation," *Int. Jnl. of Intelligent and Cooperative Information Systems (ICIS)*, 4(1), pp. 3-44, 1995.
- [3] T.D. Garvey, T.F. Lunt and M.E. Stickel, "Abductive and Approximative Reasoning Models for Characterising Inference Channels," *Proc. of the 4th Workshop on the Foundations of Computer Security*, Franconia (New Hampshire), 1991.
- [4] M.R. Genesereth and S.P. Ketchpel, "Software Agents," *Communication of the ACM (CACM)*, 37(7), 1994, pp. 48-53.
- [5] D. Jonsher and K.R. Dittrich, "An Approach for Building Secure Database Federations," *Proc. of the Very Large Database Conference (VLDB)*, pp. 24-35, 1994.
- [6] M.S. Olivier, "A Multilevel Secure Federated Database," In: J. Biskup, M. Morgenstern and C.E. Landwehr (eds), *Database Security*, pp. 183-198, 1994.
- [7] T.F. Lunt, "Aggregation and Inference: Fact and Fallacies," *Proc. of the IEEE Symposium on Research in Security and Privacy*, Oakland (California), 1989.
- [8] M. Morgenstern, "Controlling Logical Inference in Multilevel Database System," *Proc. of the IEEE Symposium on Security and Privacy*, Oakland (California), 1988.
- [9] X. Qian et al., "Detection and Elimination of Inference Channels in Multilevel Relational Database Systems," *Proc. of the IEEE Symposium on Research in Security and Privacy*, Oakland (California), 1993.
- [10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, "Object-Oriented Modelling and Design," Prentice Hall, 1991.
- [11] I. Savnik and Z. Tari, "Querying Conceptual Schemata of Object-Oriented Databases," *Int. Jnl. of Data and Knowledge Engineering (DKE)*. Under revision.
- [12] Z. Tari and P. Ratnasingham, "A Framework for Aggregation Constraint Monitoring in Federated Databases," *Proc. of the Australian Computer Science Conference (ACSC)*, Sydney, pp. 287-296, 1997.
- [13] Z. Tari and G. Fernandez, "Security Enforcement in the DOK Federated Database System," In: *Database Security X: Status and Prospects*, P. Samarati and R. Sandhu, Chapman & Hall, pp. 3-42, 1997.
- [14] Z. Tari and J. Stokes, "Designing the Regineering Service for the DOK Federated Database System," *Proc. of the IEEE Conf. on Data Engineering (ICDE)*, Birmingham, pp. 465-475, 1997.
- [15] Z. Tari, W. Cheng, K. Yetongnon, and I. Savnik, "Towards Cooperative Databases: The DOK Approach," *Proc. of the Int. Conf. on Parallel and Distributed Computing Systems (PDCS)*, Dijon, pp. 595-600, 1996.

¹NEO is the SUN CORBA-compliant product and KQML is an agent-based communication language developed by the AI research group at University Maryland Baltimore County (UMBC).

Web Implementation of a Security Mediator for Medical Databases

Gio Wiederhold

Michel Bilello

Chris Donahue

Dept. of Computer Science
Stanford University
Stanford, CA 94305

School of Medicine
Stanford University
Stanford, CA 94305

Dept. of Computer Science
Stanford University
Stanford, CA 94305

Abstract

Internet access to medical data has greatly facilitated information sharing. As health care institutions become more willing or more pressured to share some of their protected information, tools are being developed to facilitate the information transfer while protecting the privacy of the data. To this end, under the TIHI project, we have designed a security mediator, a software entity that screens both incoming queries and outgoing results for compliance with a medical institution's policies pertaining to data privacy. The system is under the control of a security officer, who enters simple rules into the system that implement the policies of the institution. In this paper, we describe the WWW implementation of the security mediator dual interface. The customer interface allows outsiders to request and receive filtered medical information from a hospital database. The security officer interface permits rule editing and resolution of cases not covered by the rule-set.

1 Introduction

The TIHI project [1, 2] has led to the design of a software system which allows secure sharing of medical information over the Internet [8]. It is designed to support interaction with collaborators, rather than to prohibit attack by foes. Therefore, it is best used in conjunction with more defensive security techniques such as public/private key systems or firewalls.

The central component of the system, the *security mediator*, is a gateway between a medical institution (e.g., a hospital), and outsiders (customers) that have a legitimate right to or interest in the institution's medical information. Typical customers include:

- Public Health Agencies
- Medical Researchers
- Community or Specialty Physicians
- Insurance Companies

The security mediator is a tool that belongs to the *security officer*, the person responsible for enforcing the medical institution's policies concerning patient data security and privacy. The security mediator helps the security officer enforce these policies by translating a security policy into a set of rules. These rules belong to three categories, depending on whether they affect the customer himself (setup rules), queries submitted by the customer (query rules), or results that follow from queries (result rules). Setup rules verify the customer's name and password, and restrict the days and times when access is allowed (i.e., a billing clerk may not be allowed to access the system on weekends). Sample query rules are Check Tables (the customer is restricted to specific tables in the database) and Check Select (the customer is limited to one *select* statement per query). The most important result rule is Check Dictionary, which checks each word contained in the results against a user-dependent dictionary to ensure that no sensitive textual information is released to the customer.

When a rule violation is detected, the query, the results, or both are sent to the security officer for review. The security officer can either approve the query as is, approve an edited form of the query, or approve a filtered set of results. Results checking is a crucial augmentation to the common model of secure access, in which no further validation is done after authentication, authorization, and certificate issue for access rights. In practice, results checking is a critical step, because the organization of the records in an institution is structured to deal with efficient local use, not with the secure matching of categories to external access rights.

All interactions with the system are recorded in the Audit Trail database. The security officer can use the Audit Trail to fine-tune the system. For example, if a customer has been entering queries in an attempt to circumvent an access restriction, the security officer can force all of the customer's queries to be reviewed

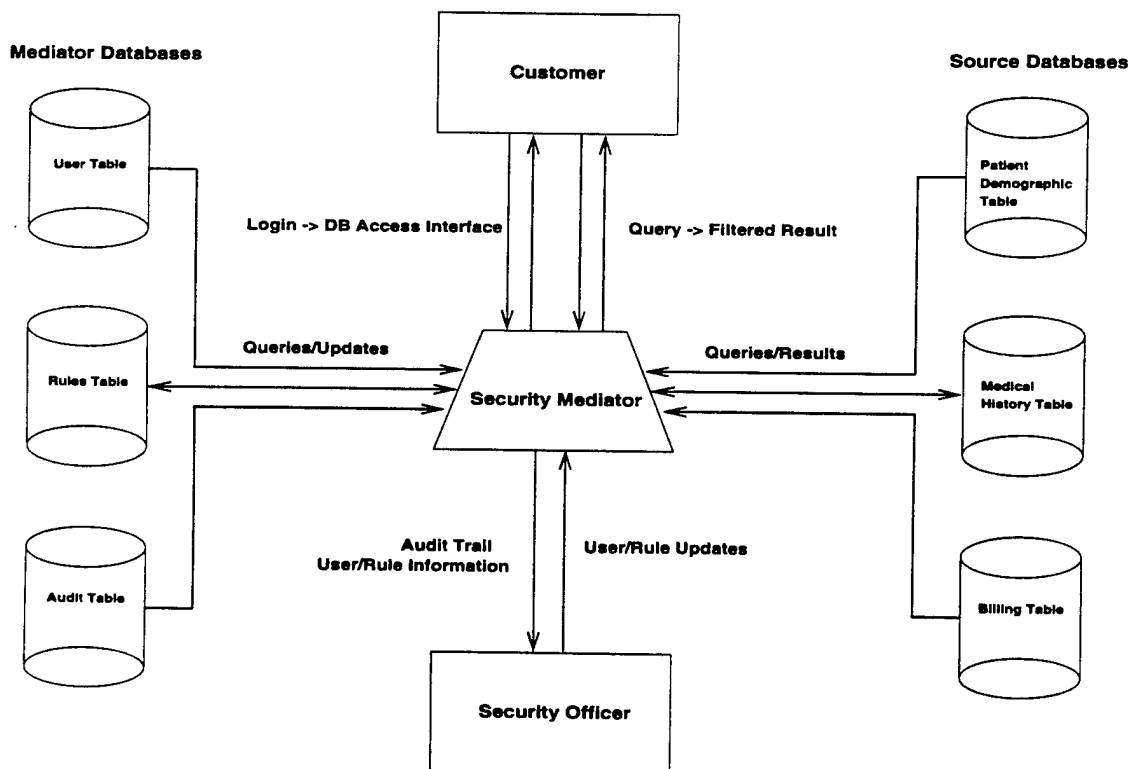


Figure 1: Overall Architecture.

manually. On the other hand, if a large number of safe queries get bumped to the security officer unnecessarily, he can relax the rules to allow the queries to pass without manual review. Each clique's dictionary can be incrementally constructed as well, with words being added as the results are manually approved by the security officer.

The subsequent sections give details of the system architecture and the WWW implementation.

2 Overall Architecture

The security mediator regulates access to database information by screening customers, queries *and* contents of results. The overall architecture is described below and diagrammed in Figure 1.

The back-end of the system is a source database containing the information that the customers are interested in. Tables from this database could include a Patient Demographics Table, a Medical History Table, and a Billing Table. This information resides on a central computer which can only be accessed by authorized personnel inside the medical institution. Therefore, the machine need not be multi-level secure.

Another component of the system is the mediator database, which stores the User Table (containing the

usernames and passwords of registered customers), the Rules Table (containing the policy rules that govern login, query, and result screening), and the Audit Table (a record of all transactions, including date, time, user identification, queries, results, and possible rule violation statements). This database typically resides on a Unix workstation protected by a multi-level security system.

The mediator engine sits on the Unix station described above. It consists of a collection of executable routines and scripts that work in concert to access the mediator and source databases in response to customers' queries or to the security officer's input.

Communicating with the security mediator engine are the Web-based customer and security officer interfaces. The customer interface allows customers to submit queries and to retrieve results from remote sites which run any operating system that supports a WWW connection. The security officer interface permits rule updates and audit trail look-ups. The security officer need not operate from the Unix station that holds the mediator engine and database.

3 Implementation

The current version of the TIHI prototype has been implemented using HTML forms and CGI scripts to connect the front-end Web interfaces with the internal databases. The architecture consists of four layers: an HTML forms user interface, Perl CGI scripts, C routines, and embedded SQL database functions. Details can be found in Figure 2. The interfaces for both the customer and the security officer are Web-based, and accessible from any browser.

3.1 Customer Interface

The customer interface consists of three screens: the login screen, the custom medical database access screen, and the result screen. Controlling the access and result screens are three Perl scripts: the Login Processor, the Query Processor, and the Result Processor.

The Login Processor reads the username, clique (membership group), and password from the login screen (Figure 3), and retrieves from the Rules Table the setup rules associated with the customer's clique. It then cycles through the relevant rules, calling each rule's corresponding C routine. Each routine returns a Pass or Fail flag. If a rule violation is detected, the Login Processor generates a standard error screen (in HTML) and returns it to the customer. No explanation is given to the customer as to why the login failed, since, given information, the customer may be able to circumvent the rule that restricted access. If all the setup rules pass, the customer is provided with a custom database access screen, which imposes a customer-dependent type of query. For example, a billing clerk would be prompted to enter a patient ID number, not a patient name, because billing clerks need not (and probably should not) know patient names in order to perform their transactions (Figure 4). Finally, the Login Processor records a successful login entry into the Audit Table.

The customer then enters a query either in SQL or by filling out custom forms, depending on the clique. For example, members of the patient clique can only request their own record, so the query is built by the mediator using the patient's name. Medical researchers, however, are allowed to enter full SQL requests. The query, as well as information about the customer and the clique, is sent to the Query Processor via HTML forms. The Query Processor then obtains the pre-processing (query processing) rules associated with the customer's clique, and cycles through the rules in the same manner as did the Login Processor. If the query passes all relevant rules, then the results are retrieved and processed by the Result Pro-

cessor. All successful queries are recorded in the Audit Table. Unsuccessful queries are sent to the Review Queue (explained below).

Successful queries cause the mediator to retrieve the corresponding results and to screen them using the Result Processor. Post-processing (result processing) rules are retrieved from the Rule Table and applied to the results. If no rule violation occurs, the results are presented to the customer in HTML tables format. A rule violation causes the query that yielded the results to be sent to the Review Queue.

If a query is unsuccessful because of a rule violation, an entry is made in the Audit Table section called the Review Queue. The username, clique, query, and the rule broken are all stored in one entry of the Review Queue (Figure 5). The Security Officer can examine each entry and decide whether the query should be allowed. The Security Officer has the option of editing the query or rejecting it altogether. In the former case, the security officer edits either the query or the results (or both), and sends the results via e-mail to the query issuer (Figure 6). Otherwise, the customer is notified via e-mail that the request was rejected.

3.2 Security Officer Interface

The Security Officer enforces the security policies of the medical institution using the TIHI system. She builds cliques and rule-sets, monitors system usage, and approves or rejects queries and results that the Security Mediator disallowed.

The Security Officer HTML interface main page gives the Security Officer a choice of six functions which can be divided into two categories: system monitoring and general maintenance.

System Monitoring:

- **Edit Results:** The Security Officer can edit unacceptable results of queries in the Review Queue, and either send the filtered results to the customer or reject the request altogether.
- **Edit Query:** The Security Officer can either edit unacceptable queries and send the results to the customer, or reject the request altogether.
- **Edit Dictionary:** The Security Officer can add to or delete words from each clique's dictionary. The Edit Clique and Edit Dictionary functions, used in conjunction with the Review Queue, allow the Security Officer to refine a clique's rule-set and dictionary in response to the results being requested.

3 Implementation

The current version of the TIHI prototype has been implemented using HTML forms and CGI scripts to connect the front-end Web interfaces with the internal databases. The architecture consists of four layers: an HTML forms user interface, Perl CGI scripts, C routines, and embedded SQL database functions. Details can be found in Figure 2. The interfaces for both the customer and the security officer are Web-based, and accessible from any browser.

3.1 Customer Interface

The customer interface consists of three screens: the login screen, the custom medical database access screen, and the result screen. Controlling the access and result screens are three Perl scripts: the Login Processor, the Query Processor, and the Result Processor.

The Login Processor reads the username, clique (membership group), and password from the login screen (Figure 3), and retrieves from the Rules Table the setup rules associated with the customer's clique. It then cycles through the relevant rules, calling each rule's corresponding C routine. Each routine returns a Pass or Fail flag. If a rule violation is detected, the Login Processor generates a standard error screen (in HTML) and returns it to the customer. No explanation is given to the customer as to why the login failed, since, given information, the customer may be able to circumvent the rule that restricted access. If all the setup rules pass, the customer is provided with a custom database access screen, which imposes a customer-dependent type of query. For example, a billing clerk would be prompted to enter a patient ID number, not a patient name, because billing clerks need not (and probably should not) know patient names in order to perform their transactions (Figure 4). Finally, the Login Processor records a successful login entry into the Audit Table.

The customer then enters a query either in SQL or by filling out custom forms, depending on the clique. For example, members of the patient clique can only request their own record, so the query is built by the mediator using the patient's name. Medical researchers, however, are allowed to enter full SQL requests. The query, as well as information about the customer and the clique, is sent to the Query Processor via HTML forms. The Query Processor then obtains the pre-processing (query processing) rules associated with the customer's clique, and cycles through the rules in the same manner as did the Login Processor. If the query passes all relevant rules, then the results are retrieved and processed by the Result Pro-

cessor. All successful queries are recorded in the Audit Table. Unsuccessful queries are sent to the Review Queue (explained below).

Successful queries cause the mediator to retrieve the corresponding results and to screen them using the Result Processor. Post-processing (result processing) rules are retrieved from the Rule Table and applied to the results. If no rule violation occurs, the results are presented to the customer in HTML tables format. A rule violation causes the query that yielded the results to be sent to the Review Queue.

If a query is unsuccessful because of a rule violation, an entry is made in the Audit Table section called the Review Queue. The username, clique, query, and the rule broken are all stored in one entry of the Review Queue (Figure 5). The Security Officer can examine each entry and decide whether the query should be allowed. The Security Officer has the option of editing the query or rejecting it altogether. In the former case, the security officer edits either the query or the results (or both), and sends the results via e-mail to the query issuer (Figure 6). Otherwise, the customer is notified via e-mail that the request was rejected.

3.2 Security Officer Interface

The Security Officer enforces the security policies of the medical institution using the TIHI system. She builds cliques and rule-sets, monitors system usage, and approves or rejects queries and results that the Security Mediator disallowed.

The Security Officer HTML interface main page gives the Security Officer a choice of six functions which can be divided into two categories: system monitoring and general maintenance.

System Monitoring:

- **Edit Results:** The Security Officer can edit unacceptable results of queries in the Review Queue, and either send the filtered results to the customer or reject the request altogether.
- **Edit Query:** The Security Officer can either edit unacceptable queries and send the results to the customer, or reject the request altogether.
- **Edit Dictionary:** The Security Officer can add to or delete words from each clique's dictionary. The Edit Clique and Edit Dictionary functions, used in conjunction with the Review Queue, allow the Security Officer to refine a clique's rule-set and dictionary in response to the results being requested.

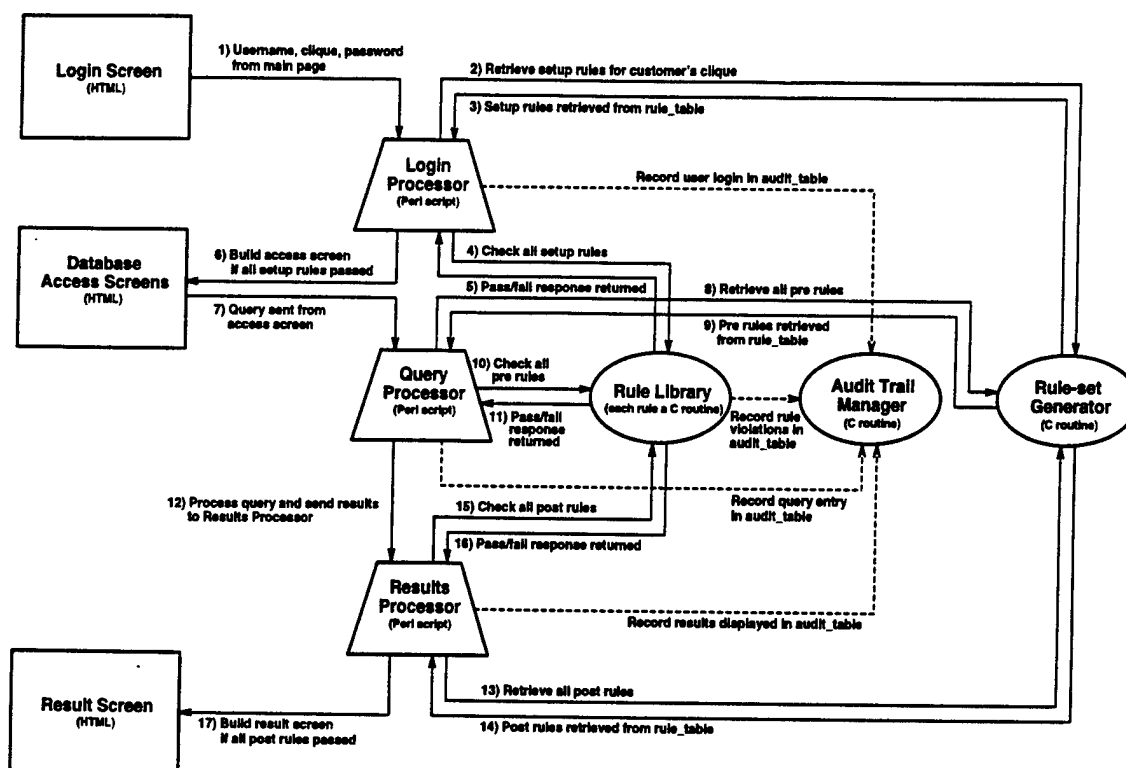


Figure 2: Details of Implementation.

- View Audit Trail: The Security Officer can make a custom query on the Audit Trail database for reporting and investigative purposes, or to improve the rule-set.

General Maintenance:

- Create Clique: The Security Officer enters the clique's name, and the names and e-mail addresses of the new clique's users. She can also choose a rule-set for the new clique from the catalogue of rules in the system.
- Edit Clique: The Security Officer can add or delete users, add to or delete rules from the clique's rule-set, or change the parameters for the active rules in the clique's rule-set.
- Edit User Database: The Security Officer can add to or delete users from the database. If a deleted user is the only member of a clique, the clique is deleted as well.
- Edit Default Rules: The Security Officer can add or delete rules, and change the parameters for the active rules in the default rule-set.

Throughout the login/query entry/result retrieval process, the activities of the customer and any intervention by the Security Mediator are recorded in the Audit Table. This information is then used by the Security Officer to generate reports and uncover suspicious trends in the use of the system. The Mediator itself uses the Audit Table to retrieve information necessary for rule application (e.g., the Last Login Time rule).

4 Future Implementation

The next generation TIHI system, which is under initial development, will differ from the current prototype in several respects.

First, the functionality of the rule-set will be increased. Instead of the static collection of rules currently used, the security officer will enjoy a dynamic rule environment. A rule compiler will be added, that will allow the Security Officer to construct rules. For example, suppose a Billing Clerk should have different access rights depending on the time of day. The Check Times, Check Tables, and Check Columns rules would be combined to create a rule that would give the Billing Clerk access to a particular set of tables and columns before 5 pm, and to a smaller set after 5 pm.

Another possible improvement would be to port the entire system to Java. A Java environment would allow for greater interactivity in both the customer and Security Officer interfaces. It would also simplify the underlying structure of the system, shrinking the number of layers from four to two (Java would be used both for the back-end routines that provide database access and for the front-end user interface screens that provide user input).

5 Summary and Conclusions

The TIHI system consolidates the security needs of an institution's database system, placing the burden on the Security Mediator. By moving the security element of the system from the databases themselves to the Security Mediator, we have accomplished several goals. First, we have created a solution that can manage an institution's data sources while disregarding its specific physical instantiation. By rigorously parsing queries and filtering results, the Security Mediator is able to overcome security holes found in the underlying data organization and storage. Second, the Security Mediator serves as a security policy implementation, designed to be used by institutional management rather than by database or network administrators. This high-level approach places the control of computer-based data resources in the hands of those responsible for an institution's information, not those responsible for its computers.

The Security Mediator concept is not limited to the health-care domain. It is applicable wherever there is collaboration between different user domains (either within an institution, or between institutions) and users' access rights have little or no correlation to the underlying structure of the data. Military and manufacturing domains are potential future test-beds for security mediator technology.

Acknowledgments

The authors would like to thank Dr. Lee Mann of Inova Health Systems, Lexical Technology, Inc., and Dr. Fred Chu for making medical data available. This work was supported in part under a subcontract of NSF grant ECS-94-22688 awarded to SRI International.

References

- [1] G. Wiederhold, M. Bilello, V. Sarathy, X.L. Qian, "A Security Mediator for Health Care Information", *Proceedings of the 1996 AMIA (formerly SCAMC) Conference*, October 1996:120-124.
- [2] G. Wiederhold, M. Bilello, V. Sarathy, X.L. Qian, "Protecting Collaboration", *Proceedings of the NISSC 1996 National Information Systems Security Conference*, Baltimore, MD, October 1996:561-569.
- [3] D.R. Johnson, F.F. Sayjdari, J.P. Van Tassel, "Missi security policy: A formal approach", Technical Report R2SPO-TR001, National Security Agency Central Service, July 1995.
- [4] B. Braithwaite, "National health information privacy bill generates heat at SCAMC", *Journal of the American Informatics Association*, 1996:3(1):95-96.
- [5] M. Hardwick, D.L. Spooner, T. Rando, K.C. Morris, "Sharing Manufacturing Information in Virtual Enterprises", *Comm. ACM*, 1996:39(2):46-54.
- [6] P.P. Griffiths, B.W. Wade, "An Authorization Mechanism for a Relational Database System", *ACM Transactions on Database Systems*, 1976:1(3):242-255.
- [7] M. Schaefer, G. Smith, "Assured Discretionary Access Control for Trusted RDBMS", In *Proceedings of the Ninth IFIP WG 11.3 Working Conference on Database Security*, 1995:275-289.
- [8] T.C. Rindfleisch, "Confidentiality, Information Technology, and Health Care", In *Communications of the Association of Computing Machinery*, Report SMI-97-0663, February 1997.
- [9] *For the Record: Protecting Electronic Health Information*, National Research Council, National Academy of Sciences, March 1997.


• Netscape: TIHI Mediator

File Edit View Go Bookmarks Options Directory Window Help

Location: <http://62.stanford.edu:8080/>

Search Now! Search Library Annotations Web Search People Software

TIHI Mediator



Access Interface

In order to access data from the hospital medical database, please provide the information requested below:

First Name:

Last Name:

Password:

Please check your status and fill relevant fields:

☒ Patient

☒ Physician/Nurse

☒ Billing Clerk

☒ Researcher --> select your research group

Please press or

Please email your comments to security-officer@hospital

Figure 3: Login Screen.

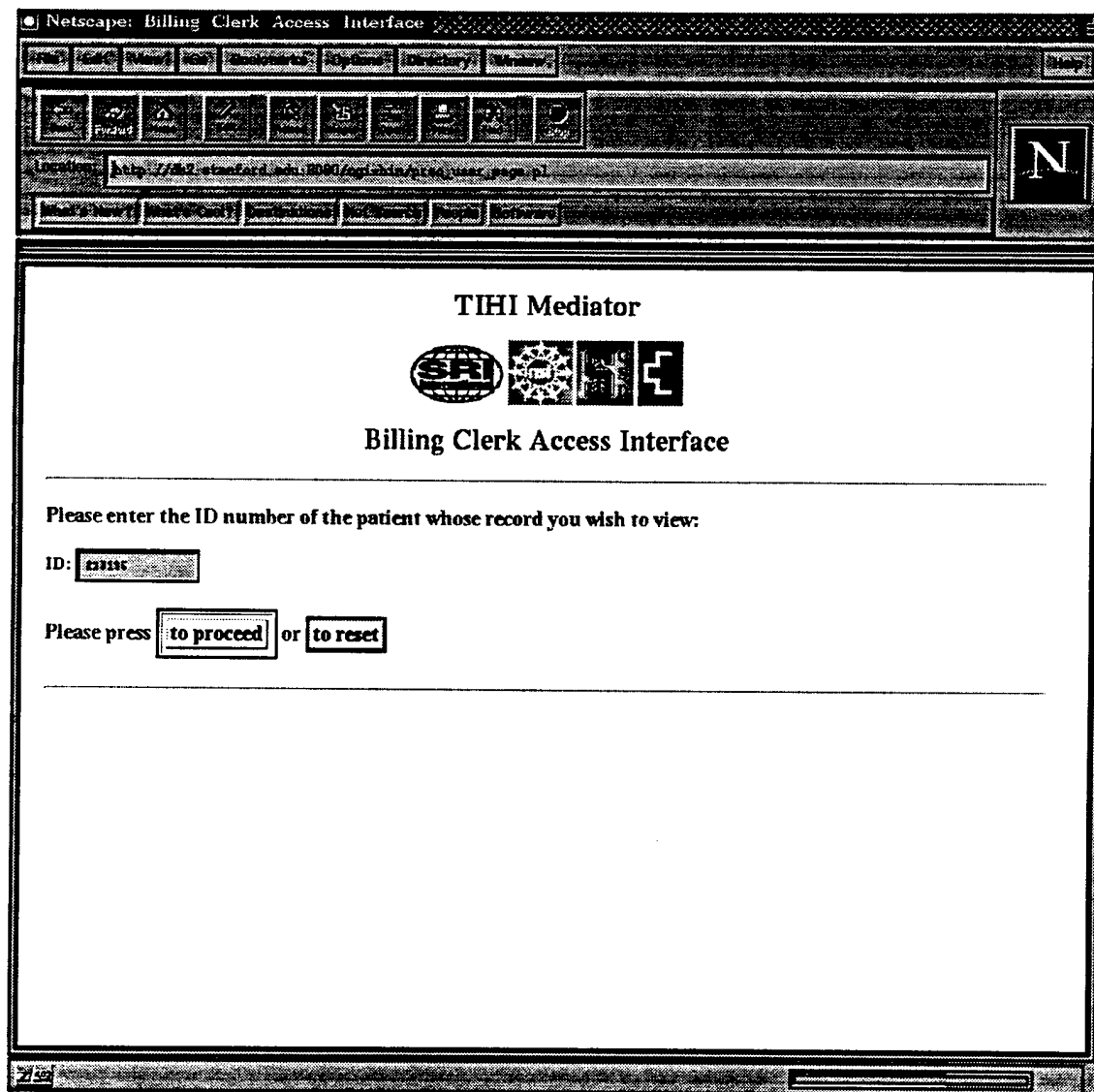


Figure 4: Custom DB Access Screen.

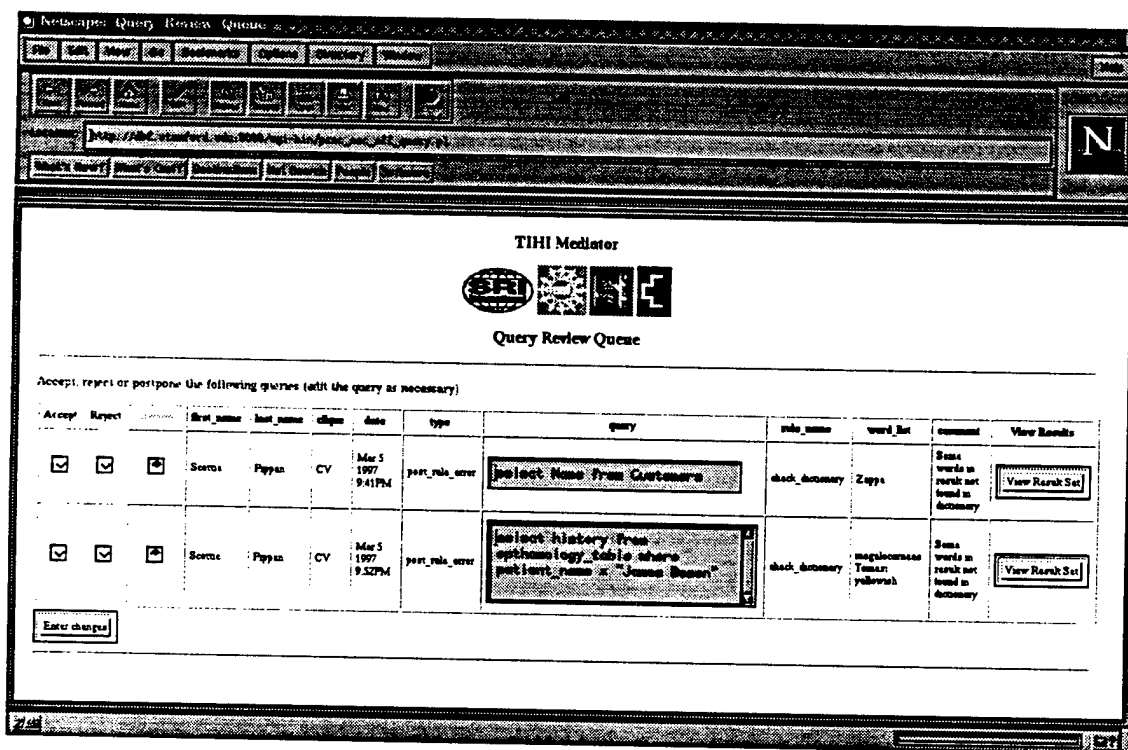


Figure 5: Review Queue.

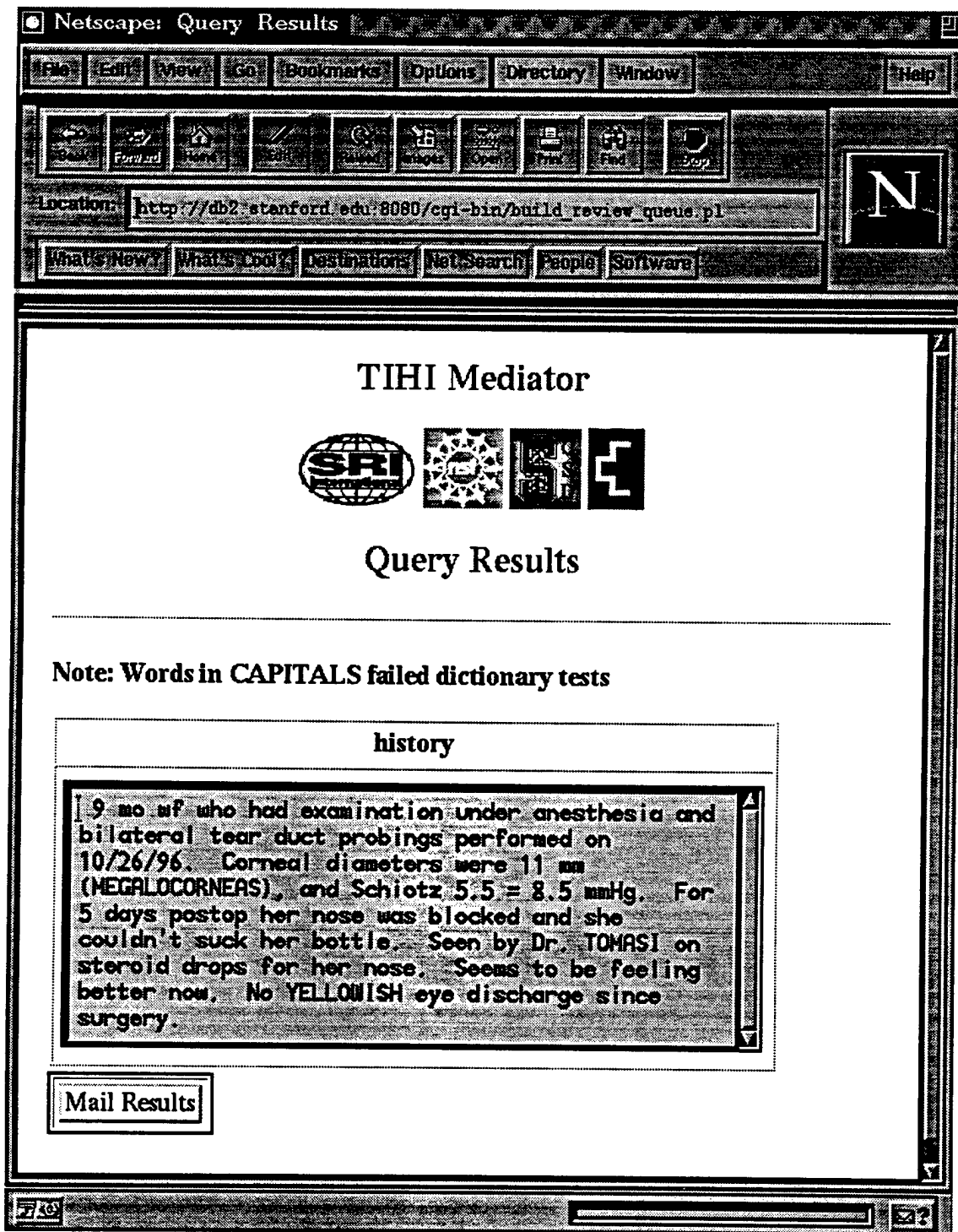


Figure 6: Results Editing Screen.

Supporting the Requirements for Multilevel Secure and Real-time Databases in Distributed Environments

Craig Chaney and Sang H. Son

Department of Computer Science, University of Virginia, Charlottesville, VA 22903
son@virginia.edu

Abstract

Conflicts in database systems with both real-time and security requirements can sometimes be unresolvable. We attack this problem by allowing a database to have partial security in order to improve on real-time performance when necessary. By our definition, systems that are partially secure allow security violations between only certain levels. We present the ideas behind a specification language that allows database designers to specify important properties of their database at an appropriate level. In order to help the designers, we developed a tool that scans a database specification and finds all unresolvable conflicts. Once the conflicts are located, the tool takes the database designer through an interactive process to generate rules for the database to follow during execution when these conflicts arise. We briefly describe the BeeHive distributed database system, and discuss how our approach can fit into the BeeHive architecture.

Keywords

Real-time, partial security, specification, conflicts, rules, analysis tool, deadline-miss ratio

1. Introduction

A *real-time system* is one whose basic specification and design correctness arguments must include its ability to meet its timing constraints. This implies that its correctness depends not only on the logical correctness, but also on the timeliness of its actions. To function correctly, it must produce a correct result within a specified time, called *deadline*. In these systems, an action performed too late (or even too early) may be useless or even harmful, even if it is functionally correct [16]. If timing requirements coming from certain essential safety-critical applications would be violated, the results could be catastrophic.

Traditionally, real-time systems manage their data (e.g. chamber temperature, aircraft locations) in application dependent structures. As real-time systems evolve, their applications become more complex and require access to more data. It thus becomes necessary to manage the data in a systematic and organized fashion. Database management systems provide tools for such organization. The resulting integrated system, which provides database operations with real-time constraints is generally called a *real-time database system*.

In many of these applications, security is another important requirement, since the system maintains sensitive information to be shared by multiple users with different levels of security clearance. As more and more of such systems are in use, one cannot avoid the need for integrating

them. Not much work has been reported on developing database systems that support both requirements of multilevel security and real-time. In this paper, we address the problem of supporting both requirements of real-time and security, based on the notion of partial security.

1.1 Real-time Database Systems

Real-time database systems extend the set of correctness requirements from conventional database systems. Transactions in real-time systems must meet their timing constraints, often expressed as deadlines, in order to be correct. In stock market applications and automated factories, a poor response time from the database can result in the loss of money and property. In many real-time database systems, transactions are given priorities, and these priorities are used when scheduling transactions. In most cases, the priority assigned to a transaction is directly related to the deadline of the transaction. For example, in the Earliest Deadline First scheduling algorithm, transactions are assigned priorities that are directly proportional to their deadlines; the transaction with the closest deadline gets the highest priority, the transaction with the next closest deadline gets the next highest priority, and so on. One important goal of a real-time transaction scheduler is to minimize or eliminate the number of *priority inversions* -- situations where a high priority transaction is forced to wait for a lower priority transaction to complete. As we shall see below, it is this goal that comes in conflict with security requirements.

1.2 Multilevel Secure Database Systems

Multilevel secure database systems have a set of requirements that are beyond those of conventional database systems. A number of conceptual models exist that specify access rules for transactions in secure database systems. One important such model is the Bell-LaPadula model [1]. In this model, a security level is assigned to transactions and data. A security level for a transaction represents its clearance level; for data, the security level represents the classification level. Transactions are forbidden from reading data at a higher security level, and from writing data to a lower security level. If these rules are kept, a transaction cannot gain direct access to any data at a higher security level.

However, system designers must be careful of covert channels. A covert channel is an indirect means by which a high security clearance process can transfer information to a low security clearance process [7]. If a transaction at a high security level collaborates with a transaction at a lower security level, information could flow indirectly. For example, say that transaction T_a wished to send one bit of information to transaction T_b . T_a has top secret clearance, while T_b has a lower clearance. If T_a wishes to send a "1", it locks some data item previously agreed upon. (This data item could be one that is created specifically for this covert channel by T_a .) If T_a wishes to send a "0", it does not lock the data item. Then, when T_b tries to read the data item and finds it locked, it knows that T_a has sent a "1"; otherwise, it knows that T_a has sent a "0". Covert channels may use the database system's physical resources instead of specific data items.

One sure way to eliminate covert channels is to design a system that meets the requirements of *non-interference*. In such a system, a transaction cannot be affected in any manner by a transaction at a higher security level. In other words, a subject at a lower access class should not be able to distinguish between the outputs from the system in response to an input sequence including actions from a higher level subject and an input sequence in which all inputs at a higher

access class have been removed [7]. For example, a transaction must not be blocked or preempted by a transaction at a higher security level.

1.3 Integration of Real-time and Security Requirements

The requirements of real-time systems and those of security systems are often in conflict [11]. Frequently, priority inversion is necessary to avoid covert channels. Consider a transaction with a high security level and a high priority entering the database. It finds that a transaction with a lower security level and a lower priority holds a write lock on a data item that it needs to access. If the system preempts the lower priority transaction to allow the higher priority transaction to execute, the principle of non-interference is violated, for the presence of a high security transaction affected the execution of a lower security transaction. On the other hand, if the system delays the high priority transaction, a priority inversion occurs. The system has encountered an unresolvable conflict. In general, these unresolvable conflicts occur when two transactions contend for the same resource, with one transaction having both a higher security level and a higher priority level than the other. Therefore, creating a database that is completely secure and strictly avoids priority inversion is not feasible. A system that wishes to accomplish the fusion of multi-level security and real-time requirements must make some concessions at times. In some situations, priority inversions might be allowed to protect the security of the system. In other situations, the system might allow covert channels so that transactions can meet their deadlines.

There are other factors, besides security enforcement, that could degrade the timeliness of the database system. For example, transient overload or failure of certain components could impact the system performance. However, regardless of the reason for impaired timeliness, relaxing security requirements always provide a positive impact on the system performance.

1.4 Our Approach

Our approach to this problem of conflicting requirements involves dynamically keeping track of both the real-time and the security aspects of the system performance. When the system is performing well and making a high percentage of its deadlines, conflicts that arise between security and real-time requirements will tend to be resolved in favor of the security requirements more often, and more priority inversions will occur. However, the opposite is true when the real-time performance of the system starts to degrade. Then, the scheduler will attempt to eliminate priority inversions, even if it means allowing an occasional covert channel.

Semantic information about the system is necessary when making these decisions. This information could be specified before the database became operational using a specification language. In this language, users would be able to express the relative importances of keeping information secure and meeting deadlines. Specifications in this language could then be "compiled" by a pre-processing tool. After a successful compilation, the system should be deterministic in the sense that an action must be clear for every possible conflict that could arise. This action might depend on the current level of real-time performance or other aspects of the system. Any ambiguities would be caught at compile time, causing the compilation to be unsuccessful. The compilation of the specification produces output that can be understood and used by the database system.

The problem of accomplishing the union of security and real-time requirements becomes more complicated in a distributed environment. In a distributed environment, having a single entity keep track of system performance in terms of timeliness and security for the entire global

database could be impractical for a number of reasons. Requiring transactions to report to this performance monitor after every execution could put more load on the network and have a negative impact on performance. The node that contained the performance monitor would be a "hotspot" and might introduce a performance bottleneck. These problems would be serious as the system got bigger, so such a solution would have a limited scalability. This brings up an interesting question: Is it better to have many performance monitors, each responsible for a small part of the database, or to have fewer of them, each with a larger domain? In other words, what granularity of the system should the performance monitors be responsible for? In our approach, there is a performance monitor responsible for every node. One of the issues to be addressed in a system with multiple performance monitors is how to optimize the database globally with only local knowledge. In our approach, this is accomplished through communication between performance monitors at each node.

In the next section, we describe some related works in the areas of specifying real-time and secure requirements for database systems, distributed security models, and some previous work in combining the requirements of real-time and secure database systems. Section 3 describes the partial security policy, the ideas behind the specification language, and the tool to analyze the language. In Section 4, we describe how our approach will fit into BeeHive, a distributed database system with real-time, security, quality of service, and fault-tolerance requirements. Section 5 concludes the paper with a discussion of future work.

2. Related Work

There has been much work on specifying security requirements. One approach is using the security constraint classification system [13], in which the authors classified a number of security constraints. Nine different categories are given, ranging from the simple, usual constraints to constraints that classify the database depending on the content or security level of data. Constraints can also depend on real-world events, information that has been previously released, and can classify associations between data, collections of data, and even other constraints. The current version of the tool reported in this paper does not provide such a flexible specification, but eventually, we need to support a complete security constraint classification for each application.

Several methods of specifying real-time requirements also exist. For example, a real-time specification method of activity and data graphs, is presented in [8]. The fundamental building block in this design is called an atomic activity. This specification system does employ some clever techniques to group and relate these atomic activities through graphs. An activity, which is defined as a set of computations, can be viewed as a transaction. Atomic activities are given a set of properties that include name, preconditions, postconditions, preemptability, state variables, importance level, timing constraints, resource requirements, and behavior. The activities are also given temporal properties, such as arrival time, ready time, scheduling deadline, completion deadline, execution time, starting time, and completion time. Our model for the specification of real-time properties was influenced by these methods, and is probably closest to the model for the atomic activities. However, some of the properties used in that model were not necessary in ours, and we needed to add a couple of properties not present in the atomic activity model.

There have been attempts to define security protocols in distributed, object-oriented environments. Two examples are Legion [15] and CORBA [3]. However, we are not aware of any previous attempts to satisfy *both* security and real-time requirements in a distributed, object-oriented environment. George and Haritsa studied the problem of combining real-time and security

requirements [5]. They examined real-time concurrency control protocols to identify the ones that can support the security requirement of non-interference. This work is fundamentally different from our work because they make the assumption that security must always be maintained. In their work, it is not permissible to allow a security violation in order to improve on real-time performance.

3. Specification

In this section, we first outline the approach to defining partial security. We then provide the details of specifying different rules for the database system.

3.1 Partial Security

As explained above, our approach will at times call for a violation of security in order to uphold a timeliness requirement. When this happens, the system will no longer be completely secure; rather, it will only be partially secure. One of the major research questions to be addressed is to identify quantitative partial security levels and to develop methods for making trade-offs for real-time requirements. Traditionally, the notion of security has been considered binary. A system is either secure or not. A security hole either exists or not. The problem with such binary notion of security is that in many cases, it is critical to develop a system that provides an acceptable level of security and risks, based on the notion of partial security rather than unconditional absolute security, to satisfy other conflicting requirements. In that regard, it is important to define the exact meaning of partial security, for security violations of confidential data must be strictly controlled. A security violation here indicates a potential covert channel, i.e., a transaction may be affected by a transaction at a higher security level.

One approach is to define security in terms of a percentage of security violations allowed. However, the value of this definition is questionable. Even though a system may allow a very low percentage of security violations, this fact alone reveals nothing about the security of individual data. For example, a system might have a 99% security level, but the 1% of insecurity might allow the most sensitive piece of data to leak out. For serious secure database applications, a more precise metric would be necessary.

A better approach involves adapting the Bell-LaPadula security model and blurring boundaries between security levels in order to allow partial security. In this scheme, only violations between certain security levels would be allowed. As the real-time performance of the system degrades, more and more boundaries can be blurred, allowing more security violations and reducing the number of security conflicts. Since there are less conflicts, this can improve the real-time performance of the system. Additionally, with this scheme, we can still make guarantees about the security of the data. See Figure 1 for an example. Here, we are considering a system with four security levels: top secret, secret, confidential, and unclassified. In Figure 1a, the system is completely secure. Figures 1b through 1d show systems that are partially secure, progressing from more secure to completely insecure. Solid lines between security levels indicate that no violations are allowed between the levels; dashed lines indicate that violations are allowed. For example, in Figure 1b, transactions that are at the unclassified level may have conflicts with transactions at the confidential level in accessing to unclassified data, resulting in a potential covert channel.

It is possible to combine this approach with the use of percentages to define partial secu-

rity. Then, the amount of security violations between two levels for which the boundary had been blurred would be required to fall below this percentage. The above example is really a special case of this scheme, where levels can either be 0% or 100%. Note that no guarantees can be made between levels that have been assigned a non-zero percentage. Guarantees can still be made between levels designated as allowing 0% security violations; for the other levels, database designers can use different percentages to denote their preferences on where they would rather have the potential security violations occur.

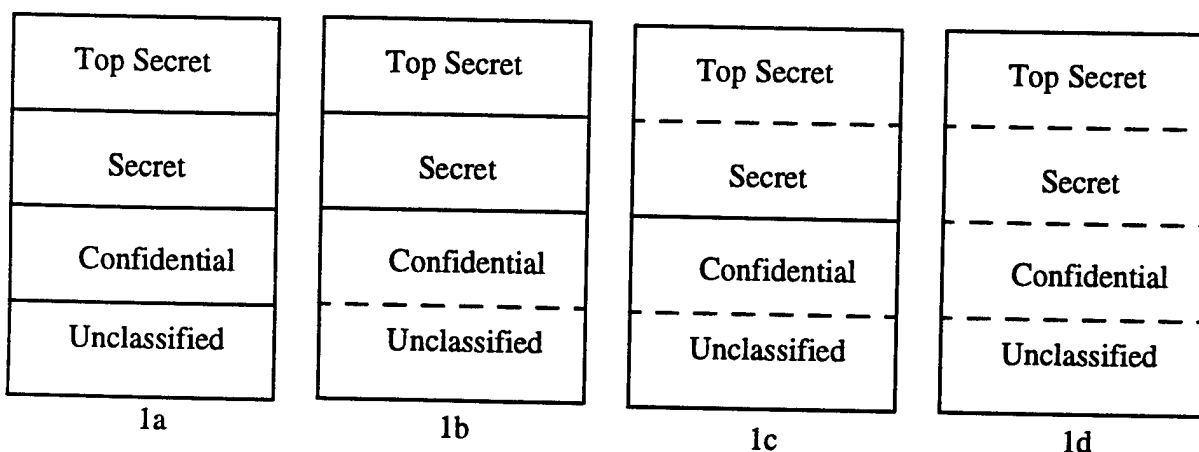


Figure 1 -- Partial Security Levels

For certain applications in which absolute security is required for safety-critical applications, any trade-offs of security for timeliness must not be allowed. The idea of partial security discussed in this paper cannot be used in such applications. Even if partial security is acceptable to an application, the system designer should be careful in identifying the conditions under which it might be dangerous to compromise the security. For example, some sort of denial of service attack could force the system into a condition where timeliness constraints are not satisfied. The system can limit the potential damage by setting up rules that can identify the situation and take appropriate actions, if necessary. For example, the system may audit the possible covert channels and log any activity that might be exploring the channel. The rules can utilize the notion of encrypted profile to either look for patterns of illegal access or, alternatively, to certify a good pattern of access.

3.2 Specification Methods

Application designers should be able to specify semantic information using a specification language to express the relative importance of keeping desired level of security and meeting timing constraint requirements. A question to be addressed in that approach is the verification of the given specification. Specifications should be compiled and verified to check any inconsistency in the requirements and to clearly determine the necessary actions to be taken. We developed a specification language that allows designers to generate rules at varying levels of detail. We have also developed a tool to analyze the specification to identify any inconsistency and produce semantic information and rules that will be maintained by the database system. The approach to specifying the security and real-time requirements is a pre-processor that aids the database designer first with locating conflicts and then with denoting their preferences according to the semantics of the data-

base. First, we will give the details of the specification language. We will then go through a few examples to illustrate the ideas.

3.2.1 Specification Language

Our specification language allows designers to create rules at varying levels of detail. In applications where much information is known about the database beforehand, designers can control security and real-time aspects of the database much more tightly than in situations where less is known beforehand or such a tight control is not required. There are three levels of detail in this specification scheme. Note that one system can use rules from all three levels if needed.

The specification consists of two parts: a *description* of the database and a set of *rules* to follow when conflicts arise. The description provides a framework for the rules. As we shall see below, the specification of both the description and the rules varies between the different levels of details. Regardless of the levels of details that are used, the first part of the specification contains facts about the database as a whole. Here, designers specify the number of data items, the number of security levels, and the number of priority levels used in the entire database.

In the first, most detailed level, designers can make rules for specific transactions. Transactions are given a number of components. Each transaction is given a readset and a writeset. These can consist of any number of data items. If no readset or writeset is given, they are assumed to be empty. The real-time requirements of a transaction are given by four variables: priority, execution time, release time, and periodicity. The periodicity of a transaction defines how often it starts executing, and the release time indicates the offset of the periodic start. Finally, transactions are given a security level.

Information about data can also be specified. Data items are specified by number, and each data item is given a security level. The specification can also contain a default security level, which is assigned to any unspecified data items. All of this information about transactions and data belong in the description portion of the specification.

Not all of these components for transactions and data items are required. In general purpose database systems, some of the information might be hard to specify. However, in many real-time applications, most information is available, since such information is necessary for schedulability analysis of the system to support the timeliness and predictability requirements. In fact, in real-time database systems, many transactions are periodic and their access pattern is known. The only truly necessary components are the security level and the priority level. If a designer leaves out, for example, the readset or the writeset, the preprocessor tool (discussed below) cannot make any assumptions about the data accessed by this transaction, so it must assume that the transaction may conflict with every other transaction.

Next, the database designer comes up with rules that define the actions that the system must take when the transactions conflict. These rules can either be static or dynamic. Static rules apply to conflicts that are resolved in the same way every time. For example, the user might specify that a conflict between two specific transactions, or two categories of transactions, will never result in a security violation.

Dynamic rules can depend on certain run-time variables that the database keeps track of during execution. Currently, dynamic rules can be based on three different dynamic variables: security violation percentage, transaction miss percentage (the percentage of transactions that have missed their deadlines), and the number of consecutive missed deadlines. Each dynamic rule has a list of clauses and a default action. A clause contains a boolean relation ($>$, $>=$, $=$, $<$, or $<=$) between one of these three dynamic variables and a constant value. Each clause also contains the

action (either violate security or violate priority) to be taken if the boolean relation is true. When a conflict is encountered by the database system, it checks the first clause. If that clause is true, it takes the associated action. If not, it checks the next clause. If none of the clauses turn out to be true, the database takes the default action. For example, a rule might be "If the security violation percentage is greater than 5, violate security. Otherwise violate timeliness." Here, the "otherwise" sentence represents the default action.

In a distributed environment, when conflict occur between transactions executing at different nodes, the action taken may need to depend on the performance at all the nodes. In that case, rules should be created that take into account the statistics on every nodes involved in the transaction. Every rule should have a "partner" rule, covering this contingency. This rule might also take into account the latency between the nodes at which the conflicting transactions are being executed.

The second level of specification detail replaces specific transactions with categories of transactions. Transactions are categorized by their security levels and priority levels. The designer can create any number of categories at any granularity that he or she feels is appropriate, and describes these categorizations in the description portion of the specification. Then, rules are created for conflicts between categories of transactions. These rules are the same as the rules for the first level.

In the third level of specification, designers create a set of rules describing actions to take in case of conflicts. Here, the conflicts are not specific; the same rule set is consulted for every conflict. Conditions would depend on the characteristics of the transactions that are conflicting or the current performance statistics. Depending on the results of the comparison, the rule would mandate either a security violation or a priority violation. All of this information belongs in the rules portion of the specification; nothing is needed in the description portion.

By carefully creating the rules, database designers can implement the partial security scheme described in the previous section. As with many other aspects of designing these rules, a tool can help designers carefully model their partial security system.

Specifications are not required to solely use one of these levels of details. The descriptions and rules for these detail levels can be mixed. In this case, when the database encounters a conflict during execution, it first searches to see if a level 1 rule applies. If not, it searches the level 2 rules, and finally checks the level 3 rules.

3.2.2 Examples

Figure 2 shows an example of a system completely specified with detail level 1. This is a small example, with only two transactions. Every relevant component of these transactions has been specified. Both transactions access data item 45, and ComputeAverage writes to it, so we have a potential conflict. Since ComputeAverage has both a lower security level and a lower priority level than SampleTransaction, this conflict cannot be resolved without causing either a covert channel or a priority inversion. Had ComputeAverage been given a higher priority than SampleTransaction, we can satisfy both requirements by allowing ComputeAverage to preempt SampleTransaction. Alternatively, if ComputeAverage had a higher security level than SampleTransaction, then both requirements could be satisfied by forcing ComputeAverage to wait for Sample transaction. As will be seen in the next section, the task of locating such conflicts can be automated.

There are two rules for this conflict -- the local rule and the non-local rule. In the rule specification, SecViolation% indicates the percentage of security violations and Trans-

Description:

```
numDataItems 5;
numSecurityLevels 4;
numPriorityLevels 4;

data[default].security = 1;
data[3].security = 2;

ComputeProfit.readset = 1, 2, 3, 4;
ComputeProfit.writeset = 5;
ComputeProfit.periodicity = 12;
ComputeProfit.priority = 3;
ComputeProfit.security = 3;

UpdatePrice.writeset = 3; # Two transactions access data item 3.
UpdatePrice.periodicity = 30;
UpdatePrice.security = 2;
UpdatePrice.priority = 2;
```

Rule for ComputeProfit-UpdatePrice conflict:

```
(SecViolation% >= 5) ~ violateTimeliness,
(TransMiss% > 10) ~ violateSecurity,
(otherwise) ~ violateTimeliness;

((LocTransMiss% <= 15) & (RemTransMiss% <= 10)) ~ violateTimeliness,
((LocSecViolation% < 10) & (RemSecViolation < 10)) ~ violateSecurity,
(otherwise) ~ violateTimeliness;
```

Figure 2 - Example of specification with fully specified detail level 1

Miss% indicates the percentage of deadline miss ratio. Each rule consists of a condition and a decision. The condition part of a rule is stated inside the parenthesis and followed by the decision after tilde (~). Conditions can be connected by logical AND (&) or OR (!). In the local rule, the first line represents a security crisis. If more than 5 percent of transactions have violated security, then this transaction cannot afford to, so it must violate timeliness. If the condition in the first line is false, the condition in the next line is checked. This line represents a real-time crisis. If more than 10 percent of transactions have missed their deadlines, then the real-time performance is suffering, so this transaction must violate security. Again, if the condition in this second line is false, the next line is checked. Here, this line is the "catch-all" rule. If none of the above rules apply, the database is instructed to violate timeliness.

The non-local rule operates in much the same way. The first line in this rule represents a state in which the real-time performance of the system is at an acceptable level either locally or at the remote site. If either condition is satisfied, the database is instructed to violate timeliness. If the real-time performance is not at an acceptable level, the system checks the second line of the rule to determine if the security of the system is acceptable. If so, it violates security; if not, it moves on to the third, "catch-all" line and violates timeliness.

Rule for SampleTransaction-ComputeAverage conflict:

```
(SecViolation% >= 5) ~ violateTimeliness,  
(TransMiss% > 10) ~ violateSecurity,  
(otherwise) ~ violateTimeliness;  
  
((LocTransMiss% <= 10) | (RemTransMiss% <= 5)) ~ violateTimeliness,  
((LocSecViolation% < 10) & (RemSecViolation < 10)) ~ violateSecurity,  
(otherwise) ~ violateTimeliness;
```

Rule for HighSecurityCategory-LowSecurityCategory conflict

```
(otherwise) ~ violateTimeliness;
```

Level 3 rules:

```
(SecViolation% < 10) ~ violateSecurity,  
(TransMiss% < 15) ~ violateTimeliness,  
(priorityLevelDifference >= 2) ~ violateSecurity,  
((TransMiss% > 10) & (SecViolation% <= 10)) ~ violateSecurity,  
(otherwise) ~ violateTimeliness;
```

Figure 3 - Example of mixed level specification

Figure 3 shows an example specification with mixed levels of detail (the database description is not shown). There are two transactions specified using detail level 1, but with only the bare minimum number of components specified. These transactions are the same as those used in figure 2. There are also a couple of transaction categories, relating to high and low security transactions. Also, there is an example of a level 3 rule set.

The rules for the conflict between the specific transactions is specified in the same manner as in the previous example. Here, we see the specification for conflicts between two transaction categories. These also are specified in the same manner.

This specific level 2 rule is also an example of a static rule -- every time that transactions in these two categories conflict, the database must violate priority and uphold security. Rules for violations between specific transactions and transaction categories can be specified, if the database designer so desires. Finally, we see a rule set for detail level 3. If none of the rules in level 1 or level 2 apply to a conflict encountered by the database, it determines the course of action by consulting this ruleset. Again, these are specified in the same manner, with the exception that a couple of new variables can be used. The variable *priorityLevelDifference* represents the difference in the priority levels of the two transactions; *securityLevelDifference* does the same for security levels.

In Figure 4 we give an example of a rule that deals with multiple conflicts. This rule is interpreted much like the level 3 rules. In the first line, the database has allowed a high number of security violations in the past, so the rule commands the database to execute the transaction with the lowest security in order to avoid all covert channels. The second line deals with a database that has allowed too many transactions to miss their deadlines; here, the database will execute the transaction with the highest priority. If the database does not have a real-time or security crisis,

then the transaction that has been waiting the longest will execute.,

3.3 Tool Implementation

When the pre-processor executes, the description portion of the specification is read and stored in internal data structures. The processor checks for syntax errors and, if no errors are found, it analyzes the specification and finds all potential conflicts between the security and real-time requirements. For completely specified level one descriptions, in order for two transactions to conflict, the following must be true:

1. They must both access the same data item.
2. At least one of the transactions must write to the data item.
3. One transaction must be at a higher security and priority level than the other.
4. The execution times of the transactions must intersect.

Every pair of transactions that satisfy these conditions are reported to the user. Of course, in less detailed descriptions, not all of these rules apply. For example, if the readset or writeset of one of the transactions is left unspecified, then the first two rules do not apply. If the timing information is incomplete for one of the transactions, the last rule does not apply. For level 2 categories, all categories might conflict, so every possible pair of categories is reported to the designer.

The user then goes through an interactive process to create rules that capture the requirement for the databases actions when these conflicts are encountered. For each conflict, the pre-processor advises the user about the implications of violating security with regard to the scheme of partial security described above. For example, in the case of a four level secure database, if a conflict occurs between transactions at the top secret level and the unclassified level, allowing a security violation would force the database into the situation of Figure 1d.

Armed with this information, the user now creates the rules for the database to follow during execution. Rules are created as explained above. The rules for detail level 3 are also inputted now. Note that since level 3 rules do not require any entries into the description portion of the specification, a database that contains rules only of level 3 will not use the description analyzer stage of the tool. Once the user has finished providing the rules, the pre-processor verifies that it can determine an action to take in any possible situation. If this is not the case, the tool finds and reports the weakness in the specification. When the specification has no remaining weaknesses, the pre-processor creates an output file that contains the choices of the user. This file will be referenced by the database during system execution.

4. Functioning in a Distributed Environment

In order to examine the distributed properties of this system further, we put it into the context of the BeeHive system, which is a distributed database system being designed with requirements beyond those of real-time and security. First, we give an overview of the BeeHive system,

```
(SecViolation% >= 10) ~ executeLowestSecurity,  
(TransMiss% > 10) ~ executeHighestPriority,  
(otherwise) ~ executeOldestTransaction;
```

Figure 4 - Example of rule for conflicts of three or more transactions.

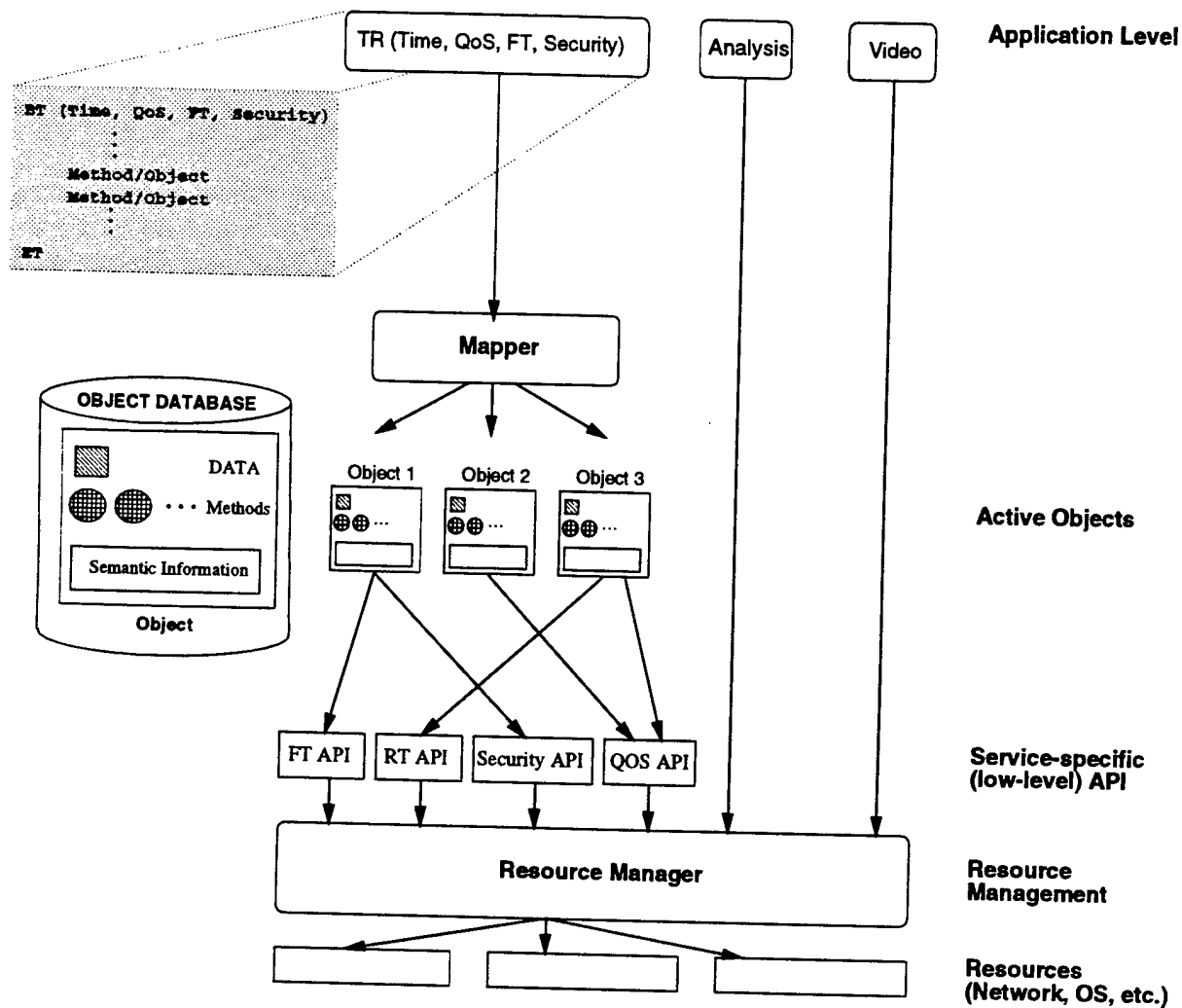


Figure 5 - Design of a native BeeHive site

and then we present how our approach fits into the BeeHive architecture. Note that we use BeeHive as one possible distributed setting to implement our approach. The actual security subsystem of BeeHive can be different from the approach we present in this paper.

4.1 BeeHive Overview

The BeeHive project at the University of Virginia[10] is an attempt to build a global virtual database with real-time, security, fault-tolerance, and quality of service. The BeeHive system is composed of native BeeHive sites, legacy sites ported to BeeHive, and interfaces to legacy systems outside of BeeHive. For the purposes of this paper, we will focus on the native BeeHive sites.

Figure 5 shows the basic design of a native BeeHive site. At the application level, users can submit transactions, analysis programs, general programs, and access audio and video data. For each of these activities the user has a high level specification interface for real-time, QoS, fault tolerance, and security. As transactions (or other programs) access objects, those objects become active and a mapping occurs between the high level requirements specification and the

object API via the mapping module. This mapping module is primarily concerned with the interface to object wrappers and with end-to-end issues. A novel aspect of the work is that each object has semantic information (also called reflective information because it is information about the object itself) associated with it that makes it possible to simultaneously satisfy the requirements of time, QoS, fault tolerance, and security in an adaptive manner. For example, the information might include rules or policies and the action to take when the underlying system cannot guarantee the deadline or level of fault tolerance requested. This semantic information also includes code that makes calls to the resource management subsystem to satisfy or negotiate the resource requirements. The resource management subsystem further translates the requirements into resource specific APIs such as the APIs for the OS, the network, the fault tolerance support mechanisms, and the security subsystem.

The resource manager of BeeHive, referred to as the "BeeKeeper", is the central entity of the resource management process. The main function of the BeeKeeper is the mapping of service-specific, possibly qualitative, QoS requirements into actual, quantitative, resource requests. The BeeKeeper contains an Admission Controller, a Resource Manager, and a Resource Allocation Module. The Admission Controller decides whether BeeHive has sufficient resources to support the requirements of a new transaction without compromising the guarantees made to currently active transactions. The Resource Allocation Module is responsible of managing the interface of BeeHive to underlying resource management systems of BeeHive components. The Resource Planner attempts to globally optimize the use of resources. The Admission Controller of the BeeKeeper merely decides whether a new application is admitted or rejected. Obviously, such a binary admission control decision leads to a greedy and globally suboptimal resource allocation. The Resource Planner is a module to enhance the admission control process and to yield globally optimal resource allocations.

4.2 Supporting Real-time and Partial Security in BeeHive

Along with the security and real-time requests of the transactions, the mapper conveys the identity of the transaction and its timestamp to each of the objects that it invokes. The timestamp is necessary to identify this specific instance of the transaction. This information is stored with the other semantic information of the object, and is conveyed to the Resource Manager through the APIs.

The real-time and security APIs allow the objects being used by transactions to convey their requirements to the resource manager. In all cases besides rules dealing with detail level 1, this is all the information about the transaction needed by the resource manager to make decisions when conflicts arise. However, in detail level 1, the resource manager needs to be aware of the identity of the transaction for which the object is executing. This information can be conveyed through either the security or the real-time API.

The admission controller will be a natural choice for the agent that detects conflicts that require the violation of either real-time or security requirements; i.e., a conflict between a high priority, high security transaction and a low-priority, low security transaction. All other conflicts are easy to resolve, and can be handled by the Admission Controller. However, for these special conflicts, the decision is delegated to the Resource Planner; this is the entity that contains and executes the rules created by the database designer.

When a transaction encounters a conflict, the Admission Controller decides (perhaps after consulting the Resource Planner) which of the two transactions is allowed to continue execution

and which must be delayed. The delayed transaction is placed in a queue associated with the executing transaction. Once that transaction is finished, the delayed transaction may begin execution. If two transactions are waiting on the queue, the Admission Controller consults the rule covering this conflict and allows one of the transactions to proceed. If more than two transactions are on the queue, and the rules do not support the execution of one of the transactions over all the other transactions, then the Admission Controller must consult the rule that deals with this situation.

The performance monitor fits best into the Resource Allocation Module. This module is closest to the resources that the statistics are representing. The feedback on resource usage that the Resource Allocation Module provides to the Resource Planner is useful for other BeeHive functions, such as for QoS and fault tolerance requirements. For our purposes, the Resource Allocation Module must keep track of the percentage of transactions that have committed a security violation or missed a deadline.

As we have seen, when conflicts occur between nodes, the action taken can depend on the performance at both nodes. Therefore, some sort of cooperation and exchange of statistics must occur between the resource managers of the nodes. In the BeeHive model, the resource managers at different nodes should communicate with each other; this will be necessary not only for our purposes, but also for the resource reservation necessary for QoS guarantees.

At first glance, this scheme seems to locally optimize the database, rather than globally optimizing it. However, when examined more closely, this node-by-node optimization may be preferable to a global optimization. Consider a database with ten nodes. In eight of these nodes, the security requirements have been upheld but the real-time performance has started to degrade. The opposite is true of the remaining two nodes. Now, a conflict occurs between these last two nodes. If the database is globally optimized, the resource managers might decide to violate security to help the overall real-time performance of the system. This decision will have little effect on the real-time performance of the eight nodes whose real-time performance is degrading, and further the security problems on the two nodes where security violations are a problem.

5. Conclusions

In this paper, we have presented mechanisms to allow the union of security and real-time requirements in distributed database systems. An important part of this union is the definition of partial security. The definition allows security violations in order to improve real-time performance, yet does not entirely compromise the security of the entire database system. However, database designers must be careful with violations between transactions whose security levels differ greatly. If a violation is allowed between transactions, say, at the highest and lowest security levels, no partial security remains in the system at all. In a system with many such conflicts, it may be very difficult to improve on real-time performance. However, it is essential that the system designer can specify how to manage the system security and real-time requirements in a controlled manner in real-world applications.

We have come up with a scheme that allows database designers to create rules at whatever level of detail that they feel is appropriate. These rules can then be analyzed by a tool, which allows designers to create a database and easily make conscious decisions about the partial security of the database. The tool can also automate the process of scanning through the complex dependencies of a database specification to find conflicts. It then informs the user of the consequences of violating security for each conflict.

Currently, we have a tool that can analyze transactions completely specified in detail level

1. This tool parses a database description, analyzes the dependencies and conflicts, and then goes through an interactive process with the user to create rules for all possible conflicts. Our future work includes extending this tool to handle rules and descriptions of levels 2 and 3. We are also developing a simulator to investigate the performance of a database that uses the output of this tool, analyzing the effects of different choices made by the user of the tool.

References

- [1] D. E. Bell and L. J. LaPadula. "Secure Computer Systems: Unified Exposition and Multics Interpretation," The Mitre Corp., March 1976.
- [2] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A. T. Rose. "MT: A Toolset for Specifying and Analyzing Real-Time Systems," Real-Time System Symposium, Lake Buena Vista, FL, December 1990.
- [3] CORBA Security, OMG Document no. 95-12-1, December 1995.
- [4] Andre N. Fredette and Rance Cleveland. "RTSL: A Language for Real-Time Schedulability Analysis," Real-Time System Symposium, Raleigh-Durham, NC, December 1993.
- [5] Binto George and Jayant Haritsa. "Secure Transaction Processing in Firm Real-Time Database Systems," ACM SIGMOD Conference, Tucson, AZ, May 1997.
- [6] T. F. Keefe, W. T. Tsai, and J. Srivastava. "Multilevel Secure Database Concurrency Control," In Proceedings of the Sixth International Conference on Data Engineering, pp 337-344, Los Angeles, CA, February 1990.
- [7] Butler W. Lampson. "A Note on the Confinement Problem," Communications of the ACM, Vol. 16, No. 10, pp 613-615, October 1973.
- [8] Alice H. Muntz and Randall W. Lichota. "A Requirements Specification Method for Adaptive Real-Time Systems," Real-Time Systems Symposium, San Antonio, TX, December 1991.
- [9] Ravi S. Sandhu and Edward J. Coyne. "Role-Based Access Control Models," IEEE Computer, vol. 29, no. 2, February 1996.
- [10] John A. Stankovic, Sang H. Son, and Jorg Liebeherr. "BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications," *Computer Science Technical Report*, CS-97-08, University of Virginia, April 1997.
- [11] S. H. Son, R. David, and C. Chaney, "Design and Analysis of an Adaptive Policy for Secure Real-Time Locking Protocol," *Journal of Information Sciences*, to appear.
- [12] Douglas A. Stuart. "Implementing a Verifier For Real-Time Systems," Real-Time Systems Symposium, Lake Buena Vista, FL, December 1990.
- [13] Bhavani Thuraisingham and William Ford. "Security Constraint Processing in a Multilevel Secure Distributed Database Management System," *IEEE Transaction on Knowledge and Data Engineering*, Vol. 7, No. 2. April 1995.
- [14] Chenxi Wang and Wm A. Wulf, "A Distributed Key Generation Technique". *Computer Science Technical Report*, CS-96-08, University of Virginia, 1996.
- [15] William A. Wulf, Chenxi Wang, and Darrell Kienzle. "A New Model of Security for Distributed Systems," *Computer Science Department Technical Report*, The University of Virginia, April 1996.
- [16] IEEE Symposium on Real-Time Technology and Applications, Montreal, Canada, June 1997.

Structured Name-Spaces in Secure Databases

Adrian Spalka and Armin B. Cremers
Department of Computer Science III, University of Bonn
Roemerstrasse 164, D-53117 Bonn, Germany
Fax: +49-228-734 382, Email: adrian@cs.uni-bonn.de

Abstract

The satisfaction of confidentiality demands in multi-level logic-based databases requires a distortion of the database's intended model. This work focuses on distortions incurred by changes in a database's name-space, ie its signature. We define extensions to a universal name-space which preserve a database's static and dynamic semantics. They are useful for achieving a high degree of confidentiality. The extensions can be embedded into the partial order of the security levels, which yields a set of hierarchical name-spaces. This hierarchy can be used to reflect both the sharing of protection units across security levels and the demands to keep them confidential.

1 Introduction

A database is generally regarded as an image of a given section of the real-world. The various elements of the database are defined in such a way that they can be identified with many important elements of the real-world section. In particular, a logic-based database can represent entities⁽¹⁾ of the real-world section, properties of entities, snapshots of the state of the properties and invariants of the snapshots. It can also track changes in the real-world section by changing the snapshots. All these aspects are covered by the theory of open logic-based databases.

A secure database must meet additional availability, integrity and confidentiality demands. The word additional indicates that a secure database is an extension to an open database and must not be defined in contravention of open databases' principles. Analogous with the conception of open databases, a secure database should represent an image of a secure real-world section.

Confidentiality demands arise in many situations of the daily life. Some are dictated by the law and others are stated by a company or a person to protect its or his interests. Referring to a particular real-world section, an explicit confidentiality demand names an element of its as the object of confidentiality and a person from whom this object should be kept confidential.

This work deals with the meaning and the enforcement of confidentiality demands whose objects of confidentiality are entities of the real-world section. It is a continuation of the work of Spalka/Cremers (1996).

In the real world adults seem to need little advice on how to keep an entity secret from another person. Limited only by their imagination there are undoubtedly countless ways. One way which a database can handle in an elegantly, relies on the naming of the entities. Let us consider a short example.

Suppose person *A* has written a report which should be kept secret from person *B*. *A* names this

1. In books on databases or on logic, eg Cremers/Griefahn/Hinze (1994) or Barwise (1991), usually the word object is used here. However, in the field of security, an object denotes a protection unit. To avoid confusion, we have decided to use the word entity instead – though we are aware it is overloaded in other fields.

report R . First of all, A hides the report from B in a safe place. As an additional protection measure A can choose a name R , which is not an element of B 's language. If B has no command of, eg, Russian or Chinese but A does, then A can name the report жалоба or 捌. Then B can neither name this report in a question nor assign the same name to one of his reports. This situation has at least two advantages. With respect to the first aspect, A will never have to lie to B about the secret report. With respect to the second, B can never – depending on the naming rules – cause a naming confusion or a naming conflict.

The purpose of this example is to manifest that differing name-spaces are a valuable means for the enforcement of confidentiality demands referring to entities. On the other hand, it is clear that this example is not intended to be directly implemented in a secure database. Well, there are database products, eg, Microsoft Access, that can work with many languages in addition to English, even with Russian and Chinese. In view of the prospective applications of secure databases, the real problem is that, though possible, it is not that simple to teach these languages to the database users.

Structured name-spaces represent a more appropriate modelling of this example. Before we introduce our approach in databases, let us take a brief look at two prominent cases in which structured name-spaces are already in use: directory-structured file systems and the handling of classified documents. In the first case, the name of a file has the form (a, b) , a is the name of a directory and b is the name of the file local to this directory. The wish to group related files and the need for a large name-space without long local names were among the original aims of adopting a directory structure. The usefulness of directories for the administration of access rights was soon realised, eg, the assignment of a home directory to each user. In the second case, the name of a classified document has also the form (a, b) . Here, a , the classification, is a security level and b is any name. The documents a person can read, create and modify are determined by comparing his clearance, which is also a security level, with the document's classification. In both cases, the components of a name (a, b) play the same roles. The name b is an element of a universal name-space \mathcal{U}^* , ie a string of characters over an alphabet \mathcal{A} that comprises the representable characters, eg, the letters of the English alphabet. Since each directory and each security level has its own name-space, \mathcal{U}^* is a local name-space. The component a represents a name-space selector. Lastly, the set of all possible names (a, b) constitutes the global universal name-space. With respect to the confidentiality of an entity with the name (a, b) , if the selector a is unavailable to a user u , then u cannot address (a, b) nor can he create an entity with the name (a, b) . Thus this constellation supports the efforts to keep the existence of (a, b) secret from u .

Let us now turn our attention to databases. First of all, we would like to stress that logic-based databases offer an excellent framework for the study of the meaning, the properties and the effects of new database elements. Logic is today a well-understood formal instrument and the notion of logic model is a succinct and elegant description of data semantics. From a practical perspective, logic-based databases comprise also relational databases – as soon as the desirable formulae are established it is easy to get down to tables and rows. Lastly, and this is very important, this approach encourages a clear separation of declarative and procedural concepts.

The name space of a logic-based database, ie its language, is determined by its signature. Specific to databases, the signature comprises an infinite set of only constant function symbols and a finite set of predicate symbols. The function symbols represent a pool out of which terms are con-

structed. The sole purpose of a ground, ie variable-free, term is to serve as a name for a particular entity of the real-world section. The set of ground terms thereby comprises the names that can be assigned to these entities. A predicate symbol is used to express a simple statement on one or more entities' property or relationship. A ground atomic formula is a concrete statement on the entities named by the formula's terms.⁽²⁾ Its truth value is determined by the database's intended model. Speaking in a more illustrative manner, the set of predicate symbols corresponds to the names of all (base and derived) relations, the set of ground terms to possible attribute values and the set of ground atomic formulae to possible tuples in the relations' extension.

This situation represents a flat name space. To refer to an entity one just has to know its name, ie the ground term that has been assigned to it; similarly, each property can be referred to by its predicate symbol. Since both the ground terms and the predicate symbols are sets, there can never be distinct entities or distinct properties that go by the same name.

Any serious database product can deny access to a relation to a user. In response to a command that refers to such a relation the database rejects the command, usually with the explanation that this relation is unknown or undefined. This behaviour is perfectly in accord with the theoretical side. We have created an individual signature for a user. Its set of predicate symbols is a subset of the open database's predicate symbols – there is no change in the function symbols. But, at present, it is not possible to restrict the function symbols for a user nor to give him a set of private function symbols. With respect to both deficits, there is no way a database will reject an update command on the grounds that a ground term is unknown or undefined. And a select command will always return an answer-set, even if it is empty.

Spalka/Cremers (1996) introduced a secure logic-based database, in which objects of confidentiality are ground atomic formulae. The authors defined the formal meaning of a confidentiality demand by characterising the difference between the open database's intended model and one that can be said to satisfy the demand. This approach retains the original open database, which corresponds to the intended state of affairs, and yields a set of deliberately falsified individual databases. It is essential to realise that this step has not created a group of new alien databases. They all still refer to the same real-world section: the original open database captures its intended image, and an individual, falsified one presents a distorted image of it.

In this work we extend that approach and impose the structure of a global universal name-space on the sets of terms and predicate symbols. The local subspaces are those of open databases. The choice and the handling of the name-space selectors is the part relevant to security. We show, firstly, that the most general case of sharing and hiding, ie when a user can share an entity with any other group of users or keep it secret from them, corresponds to choosing the users' power set as the set of name-space selectors. And, secondly, the identification of security levels with name-space selectors yields a canonical interpretation of confidentiality demands stated for terms and predicate symbols. These extensions preserve all semantic properties of open databases and all security properties of the database of Spalka/Cremers (1996).

The subsequent section introduces the fundamentals of open logic-based databases. Section 3 deals with previous works in this field. Beginning with some prominent algebraic and logical approaches, it presents a concise yet complete definition of the database of Spalka/Cremers

2. More complex statements can be expressed by forming more complex formulae.

(1996). Section 4 comprises the main results of this work. Their application is illustrated with two examples in section 5. Lastly, the conclusion summarises the main results of this work and presents a brief outlook.

2 Logic-based databases

An alphabet \mathcal{A} is a non-empty and finite set of symbols, such that any string of \mathcal{A} 's elements has a unique decomposition into \mathcal{A} 's elements. \mathcal{A}^* denotes the set of strings of finite length of elements of \mathcal{A} . Let $F = \mathcal{A}^*$ be the set of function symbols, $P \subseteq \mathcal{A}^*$ a finite set of predicate symbols, and $\rho_F : F \rightarrow \{0\}$ and $\rho : P \rightarrow \mathbb{N}_0$ the functions determining the arity of the symbols. Then $\Sigma = (\mathcal{A}, P, \rho)$ is a database signature. Let V be an infinite set of variables. Then $T(\Sigma) = F \cup V$ is the set of terms over the signature Σ . $T_G(\Sigma) = F$ is the subset of ground terms. The set of atomic formulae over Σ is $A(\Sigma) = \{p(t_1, \dots, t_n) \mid p \in P, \rho(p) = n, t_i \in T(\Sigma), i = 1, \dots, n\}$ and $A_G(\Sigma)$ is the subset of ground atomic formulae. With $\alpha \in A(\Sigma)$, α is a positive literal and $\neg\alpha$ a negative literal. The first order language over the signature Σ is the smallest set $L(\Sigma)$ with the following properties: $A(\Sigma) \subseteq L(\Sigma)$; if $\varphi, \psi \in L(\Sigma)$, then $(\varphi) \vee (\psi) \in L(\Sigma)$; if $\varphi \in L(\Sigma)$, then $\neg(\varphi) \in L(\Sigma)$; if $\varphi \in L(\Sigma)$ and $X \in V$, then $\forall X : \varphi \in L(\Sigma)$. $L_0(\Sigma)$ is the subset of closed formulae, viz, all variables of a $\varphi \in L_0(\Sigma)$ are quantified. A normal clause is a closed formula $\alpha \leftarrow \lambda_1 \wedge \dots \wedge \lambda_n$ in which all variables are assumed to be universally quantified, $\alpha \in A(\Sigma)$ is the clause's head and $\lambda_1 \wedge \dots \wedge \lambda_n$, a conjunction of literals, its body. A clause is range-restricted if any variable that occurs in the clause occurs also in a positive literal in its body. $C_R(\Sigma)$ is the set of range-restricted clauses over Σ .

Let D and Ω be sets such that there is a $\omega_F \in \Omega$, $\omega_F : F \rightarrow D$ and for each $p \in P$, $\rho(p) = n$, there is a $\omega_p \in \Omega$, $\omega_p : D^n \rightarrow \{\text{True}, \text{False}\}$. Then $M(\Sigma) = (D, \Omega)$ is an interpretation for the signature Σ . Alternative notations for restrictions to $M(\Sigma) = (D, \Omega)$: for ground atomic formulae $M_A(\Sigma) : A(\Sigma) \rightarrow \{\text{True}, \text{False}\}$; for closed formulae: $M_0(\Sigma) : L_0(\Sigma) \rightarrow \{\text{True}, \text{False}\}$. (Some works define a model as $M_A^{-1}(\Sigma)(\text{True}) \subseteq A_G(\Sigma)$). $M(\Sigma)$ is a model of Φ , $M(\Sigma) \in \text{Mod}(\Phi)$, $\Phi \subseteq L_0(\Sigma)$, if $\forall \varphi \in \Phi : M_0(\Sigma)(\varphi) = \text{True}$. $M(\Sigma) = (D, \Omega)$ is a Herbrand-interpretation or a Herbrand-model if $D = T_G(\Sigma)$ and $\omega_F = \text{id}$. $\Psi \subseteq L_0(\Sigma)$ is a logical consequence of $\Phi \subseteq L_0(\Sigma)$, $\Phi \models \Psi$, if $\text{Mod}(\Phi) \subseteq \text{Mod}(\Psi)$.

We assume that the intended semantics of a set $\Phi \subseteq L_0(\Sigma)$ is defined by its completion⁽³⁾ with respect to Σ , $\text{comp}(\Phi, \Sigma)$.

Let Σ be a database signature, $C \subseteq L_0(\Sigma)$, $\text{Mod}(C) \neq \emptyset$, a finite set of closed formulae over Σ and $I \subseteq C_R(\Sigma)$ a finite set of safe clauses over Σ such that $\text{comp}(I, \Sigma) \models C$. Then $D = (\Sigma, C, I)$ is a logic-based database with completion semantics. Σ defines the database language, C is the set of integrity constraints, I is a valid present state and the intended semantics of I is defined by the unique Herbrand-model $M(\Sigma) = (D, \Omega)$ of $\text{comp}(I, \Sigma)$.

A transaction τ – in a declarative notion – is a partitioned set $\tau = \delta \cup \iota$, $\tau \subseteq A_G(\Sigma)$. The application of τ to $M(\Sigma)$ yields an interpretation $M'(\Sigma)$, such that: $M(\Sigma)(\delta) = \text{True}$ and $M'(\Sigma)(\delta) = \text{False}$, $M(\Sigma)(\iota) = \text{False}$ and $M'(\Sigma)(\iota) = \text{True}$, and $M(\Sigma)(\alpha) = M'(\Sigma)(\alpha)$ for any $\alpha \in A_G(\Sigma) \setminus \tau$. If $M'(\Sigma) \in \text{Mod}(C)$, then τ is accepted and $M'(\Sigma)$ becomes the present model of D ; otherwise τ is rejected. τ is a singleton-update if $|\tau| = 1$.

3. Cf, eg, Das (1992) or Cremers/Griefahn/Hinze (1994).

3 Previous works

3.1 Algebraic approaches

Most algebraic approaches are based on the multi-level relational data model introduced by Denning et al (1987). This model does not mention name-spaces explicitly and a formal definition of its signature yields only a flat name space. However, the model leaves the impression that the assignment of access classes to the elements of a tuple also intends to structure the name space.

Various problems of this model have already been observed by its inventors. Those related to the name space in particular have been aptly stated by Gajnak (1988). The author investigates the adaptability of the entity-relationship modelling to multi-level security requirements and identifies three fundamental principles of multi-level databases which must not be violated⁽⁴⁾. The important semantic determinacy principle states that '... factual dependencies should be non-ambiguous'⁽⁵⁾. This property is violated by SeaView's treatment of polyinstantiation and the author gives an example in which polyinstantiation can mean that: one database entry is an alias for another one, a secret entry has been leaked or the two entries refer to two real world objects. He concludes aptly that in this situation referential integrity as such must be ambiguous. Regrettably, the author's final advice – which we strongly support – that '... the determinacy principle should be supported directly by multi-level secure data models'⁽⁶⁾ has been given little attention in the following years.

Though implicitly, Sandhu/Jajodia (1992a) also deal with naming conflicts in SeaView. The work proposes to use polyinstantiation for cover stories only.⁽⁷⁾ For the implementation of a cover story the authors use the special value, 'restricted', which was introduced in Sandhu/Jajodia (1990) and Sandhu/Jajodia (1992b). The authors demand that primary keys must not be polyinstantiated, ie, a primary key is unique regardless of its access class. Three ways to achieve this are suggested: 'Make all keys visible'⁽⁸⁾, 'Partition the domain of the primary key'⁽⁹⁾ and 'Limit insertions to be done by trusted subjects only'⁽¹⁰⁾. The first suggestion says that a flat name space is fine if confidentiality demands for function symbols are not admitted. The second one would violate the name space's property of being a universal one. And the third one shifts the problem of name spaces to that of update rights.

3.2 Logical approaches

Among the first works which – although implicitly – consider non-uniform name-spaces are Thuraisingham (1991) and Thuraisingham (1992). Based on the conviction that standard logic is inadequate, the author attempts to formalise the rules and properties of mandatory access control models in NTML, a non-monotonic logic.

Although NTML has been shown to be not sound⁽¹¹⁾, Garvey et al (1992) present a similar idea

4. Gajnak (1988):189.

5. Gajnak (1988):183.

6. Gajnak (1988):189.

7. We note that this is an assumption contrary to the one made in, eg, Sandhu/Jajodia (1991).

8. Sandhu/Jajodia (1992a):315.

9. Sandhu/Jajodia (1992a):315.

10. Sandhu/Jajodia (1992a):316.

11. Garvey et al (1992):160.

of hiding function symbols. This work introduces a multi-level database as a collection of databases. The data of each database is a theory of the function-free subset of first-order logic.⁽¹²⁾ An access class can be assigned to a function symbol, a predicate symbol or a whole fact. In the first two cases, the structure of access classes induces an analogous structure of first-order languages. However, this does not hold for the third case if the database is polyinstantiated.⁽¹³⁾ The main problem of this view is that the idea to keep a function symbol secret is not examined with respect to its compatibility with the usual database modelling approach, which assumes a universal name space. Promising though the authors' observations are, until today the approach has not been further developed.

3.3 Secure logic-based databases with a common name-space

Spalka/Cremers (1996) present an axiomatic interpretation of security-level-based mandatory security policies in logic-based databases that establishes database properties as proofs from only a few assumptions. Since confidentiality demands can be stated only for ground atomic formulae, the authors define a secure logic-based databases with a common name-space. One of its important properties is the independence of the semantics from any particular confidentiality demands, ie, the addition or removal of confidentiality demands does not affect the semantics of the data nor the notion of integrity. The database of Spalka/Cremers (1996) is defined as follows.

$MCP = (U, O, (S, \leq), G)$ is an instance of a security-level-based mandatory confidentiality policy such that: U is a set of individuals; O is a set of protection units; S is a set of security levels on which a partial order ' \leq ' is defined; $G: U \cup O \rightarrow S$ is a labelling-function that assigns a security level to each individual and protection unit; and with respect to the legal obligation that 'The dissemination of information of a particular security level (including sensitivity level and any compartments or caveats) to individuals lacking the appropriate clearances for that level is prohibited by law'⁽¹⁴⁾, the following Primitive Mandatory Requirement is satisfied: $o \in O$ should be kept secret from $u \in U$ if $G(o) \leq G(u)$ does not hold.

Let $D = (\Sigma, C, I)$ be an open database with the intended model $M(\Sigma)$. For each $s \in S$ there is a database $D_s = (\Sigma_s, C_s, I_s)$ with a unique intended Herbrand-model $M^s(\Sigma_s)$ for Σ_s . $M(\Sigma)$ is the image of the open world-section and any $M^s(\Sigma_s)$ is a distortion thereof. The distortion of $M^s(\Sigma_s)$ with respect to $M(\Sigma)$ is recorded in its distortion-log $P_s = (A_s, X_s, Z_s)$. P_s is a partition of $A_G(\Sigma)$ such that: $\forall \alpha \in A_s : M_A^s(\Sigma_s)(\alpha) = M_A(\Sigma)(\alpha)$, $\forall \alpha \in X_s : M_A^s(\Sigma_s)(\alpha) = \neg M_A(\Sigma)(\alpha)$ and $\forall \alpha \in Z_s : \alpha \notin A_G(\Sigma_s)$. The common name-space is due to the assumption that $\Sigma_s = \Sigma$ for all $s \in S$, ie $P_s = (A_s, X_s, \emptyset)$.

A unit of protection can be any ground atomic formula $\alpha \in A_G(\Sigma)$. The intended meaning of the primitive mandatory requirement for ground atomic formulae is as follows. Let $\alpha \in A_G(\Sigma)$ and $s \in S$. If $G(\alpha) \leq s$ does not hold and $\alpha \in A_G(\Sigma_s)$, then $M_A^s(\Sigma)(\alpha) = \neg M_A(\Sigma)(\alpha)$, ie $\alpha \in X_s$.

A user $u \in U$ can query and update any database D_s with $G(u) \geq s$. $\tau_{u,s} = \delta \cup \iota$, $\tau \subseteq A_G(\Sigma_s)$,

12. At this point, however, the authors do not state how to find the intended model of such a theory. They realise that a straight application of the Closed World Assumption may lead to a contradiction. To omit this problem, they propose to view the theory as a set of beliefs, yet without specifying the intended model.

13. Garvey et al (1992):160.

14. Landwehr (1981):249.

is a transaction submitted by u and applied to D_s , viz, to $M^s(\Sigma_s)$.

The powers of $u \in U$ are expressed as update-rights $R_{G(u)} \subseteq A_G(\Sigma_{G(u)})$: $\tau_{u,s}$ is rejected if $\tau_{u,s} \cap R_{G(u)} \neq \tau_{u,s}$. Congruent with the meaning of powers, the assumption Observance of Powers states that the database is not allowed to ignore a transaction a user is entitled to execute, ie $R_{G(u)} \subseteq A_{G(u)}$. This – presumably common-sense assumption – has two important implications. The first one is the Update Truthfulness lemma. Suppose that the application of the transaction $\tau_{u,s}$, $s = G(u)$, to $M^s(\Sigma_s)$ yields $\bar{M}^s(\Sigma_s)$. If $\tau_{u,s}$ is accepted by D_s , then it is also propagated to D , ie, it is also applied to D and the following condition holds: $\forall \alpha \in \tau_{u,s} : \bar{M}^s(\Sigma_s) = \bar{M}(\Sigma)$. The second one is the Subordinate Validity lemma. If $\tau_{u,s}$ is accepted by D_s , then it is also accepted by D .

Lastly, the assumption is made that the database need not be a user's only source of information on the real-world section. For a user $u \in U$ the set $K_s \subseteq A_G(\Sigma_s)$, $s = G(u)$, expresses u 's assumed special knowledge on the real-world section. Informally, K_s comprises those facts the truth value of which u can inspect without consulting the database. Formally, it is the condition that $\forall \alpha \in K_s : M_A^s(\Sigma_s)(\alpha) = M_A(\Sigma)(\alpha)$. An immediate consequence of it is the Observance of Knowledge lemma: $K_{G(u)} \subseteq A_{G(u)}$ for any $u \in U$.

The satisfaction of confidentiality demands is not unconditional. The database is required to ascertain beforehand the circumstances of several events. Just as any transaction will be rejected if integrity is violated, a confidentiality demand will be rejected if any of the above-defined invariants is violated. The success depends on the ability to find a distortion of the affected models that respects these invariants. Given a particular confidentiality demand, the enforcement method used by Spalka/Cremers (1996) relies on aliases, ie, on the additional reversal of one or more facts' truth values.

4 Structured and hierarchical name-spaces

In an open logic-based database the signature $\Sigma = (\mathcal{A}, P, \rho)$ defines a flat name-space. It is not possible to remove from it nor to add to it single ground terms without destroying its universal name-space property. Our approach preserves this property on a local basis by replacing the flat name-space with a global name-space that comprises several local universal name-spaces.

As outlined in the introduction, an element of the global name-space is a tuple (a, b) such that a designates a local name-space and b is an element of the local name-space. The formal extension is straightforward. Given a universal name-space \mathcal{Q}^* and a set E , we define $N = E \times \mathcal{Q}^*$ to be a global universal name-space. E is a prefix-set which comprises the names (or selectors) of the local name-spaces, and $(a, b) \in E \times \mathcal{Q}^*$.

The usefulness of this extension with respect to a particular application depends on a suitable choice of the prefix-set and on the allocation of name-space selectors to the application's users. We first take a look at the most general situation – it can be subsequently tailored to any special requirements concerning the sharing and the confidentiality of the global name-space's elements.

Let U be a set of users and \mathcal{Q}^* a universal name-space. Firstly, set $E = \mathcal{P}(U)$, the power-set of U . Then for each $e \in E$ $N_e = \{e\} \times \mathcal{Q}^*$ represents a local universal name-space. Secondly, assign to each user $u \in U$ a set $E_u \subseteq \{e \in E \mid u \in e\}$. And, lastly, define u 's name-space, N_u , as

$$N_u = \bigcup_{e \in E_u} N_e$$

Then

$$N = \bigcup_{u \in U} N_u$$

is the application's global universal name-space.

There are two main reasons for this definition.

The first one is the possibility of using the name-space selectors for the expression of both the separation and the sharing of name-spaces among users. Let $e \in E$. Then the entities designated by the elements of $N_e = \{e\} \times \mathcal{Q}^*$ are supposed to be shared among the users $u \in e$ and kept secret from the users $u \in U \setminus e$. For example, if the user $u \in U$ should have an exclusive private local name-space, then we add the selector $\{u\} \in E$ to his set of name-space selectors E_u . If there should be a name-space shared by the group of users u_1, \dots, u_k , then we add the selector $\{u_1, \dots, u_k\} \in E$ to all E_{u_1}, \dots, E_{u_k} . Setting $E_u = \{e \in E \mid u \in e\}$ gives the maximum flexibility. Here, each user has a private name-space and shares a name-space with any other group of users.

The second reason is the canonical partial order defined by the inclusion relation on a power set. It subsumes any partial order on the set's elements in the following sense⁽¹⁵⁾. Let (S, \leq) be a partial order. Then there is a unique subset $E \subseteq \mathcal{P}(S)$ such that (S, \leq) and (E, \supseteq) are isomorphic, ie, there is a bijective function $h : S \rightarrow E$ such that $s_1 \leq s_2 \Leftrightarrow h(s_1) \supseteq h(s_2)$, $s_1, s_2 \in S$, and $s \in h(s)$.

On these grounds we can express the extension to the database defined in section 3.3 in a very succinct way. Let $MCP = (U, O, (S, \leq), G)$ be the given instance of a security-level-based mandatory confidentiality policy. The main step is to find the partial order (E, \supseteq) isomorphic to (S, \leq) . Then define the open database $D = (\Sigma, C, I)$, $\Sigma = (N, P, \rho)$ with the global universal name-space $N = E \times \mathcal{Q}^*$, $P \subset N$ and $\rho : P \rightarrow N_0$. With every security level $s \in S$ there is an associated database $D_s = (\Sigma_s, C_s, I_s)$. The name-space of D_s is a collection of local universal name-spaces within N . Set, as shown above, $E_s = \{e \in E \mid s \in e\}$. Then set $\Sigma_s = (N_s, P_s, \rho_s)$ with $N_s = \bigcup_{e \in E_s} N_e$, $P_s = \{p \in P \mid p \in N_s\}$ and $\rho_s = \rho|_{P_s}$.

We can eliminate (E, \supseteq) in a simple transformation. The result is a definition that refers only to the security levels S . For the function $h : S \rightarrow E$ is bijective, we can define $D = (\Sigma, C, I)$ and $\Sigma = (N, P, \rho)$ with $N = S \times \mathcal{Q}^*$, $P \subset N$ and $\rho : P \rightarrow N_0$. In the next step, we replace the inclusion with the partial order on S . Let $B_s = \{s' \in S \mid s' \leq s\}$. Then set $N_s = B_s \times \mathcal{Q}^*$, $P_s = \{p \in P \mid p \in N_s\}$ and $\rho_s = \rho|_{P_s}$, and define $\Sigma_s = (N_s, P_s, \rho_s)$ and $D_s = (\Sigma_s, C_s, I_s)$.

Lastly, we define the security semantics of the extension, ie, the meaning of a confidentiality demand stated for ground terms or predicate symbols. A unit of protection can be any ground term $t \in T_G(\Sigma)$ and any predicate symbol $p \in P$. The intended meaning of the primitive mandatory requirement for ground terms and predicate symbols is as follows.

Let $w = (a, b)$ be a ground term or a predicate symbol and $s \in S$. If $G(w) \leq s$ does not hold,

15. Cf Davey/Priestley.

then $w \notin T_G(\Sigma_s)$ and $w \notin P_s$, ie $w \in Z_s$.

We immediately see that a necessary condition for the satisfaction of a confidentiality demand for $w = (a, b)$ is $a \geq G(w)$.

To conclude the theoretical part, we note that in our database – like in every logic-based database with a unique intended model – there are no NULL-values. There are good reasons for their exclusion.⁽¹⁶⁾ Speaking in a colloquial manner, NULL-values make the semantics of even open databases messy. For example, just by looking at a database there is no way to tell if a NULL-value means that a value is unknown or if the attribute is not applicable here. Although we have not investigated it formally, the corresponding works on open databases indicate that their use for confidentiality purposes will not make the situation any less confusing.

Let us briefly restate the semantic properties of this database. The open database $D = (\Sigma, C, I)$ and its intended model $M(\Sigma)$ capture at all times the image of the open real-world section. If there is a user from whom nothing is kept secret, then this is his database. The database at a security level $s \in S$, $D_s = (\Sigma_s, C_s, I_s)$, and its intended model $M^s(\Sigma_s)$ capture an image of the same open real-world section as $D = (\Sigma, C, I)$. Due to confidentiality demands, this image can be distorted. Speaking somewhat loosely, the distortion-log $P_s = (A_s, X_s, Z_s)$ keeps track of the – deliberately introduced – lies, X_s , and the – also deliberately withdrawn – missing information, Z_s . A query submitted to D_s is always evaluated with respect to $M^s(\Sigma_s)$. This means that no trusted user is ever confused⁽¹⁷⁾, we do not have to guess about versions, degrees of interest or recency⁽¹⁸⁾, nor do we have to deal with imprecise beliefs⁽¹⁹⁾. The acceptance of update operations, which can be combined with the specification of confidentiality demands, is subject not only to the integrity constraints but also to the powers of the issuing user, the Observance of Powers assumption and the Observance of Knowledge lemma. They all must always be satisfied. Aliases are additional distortions a user can introduce to support his confidentiality demand.

5 Two modelling examples

5.1 A small transport company

Let us illustrate the application of our theory in an example of a company that transports goods in cars.

Suppose that the alphabet \mathcal{A} comprises the letters of the English alphabet and the digits 0 through 9, and that there are four security levels $L = \{l_1, l_2, l_3, l_4\}$ such that $l_1 > l_2 > l_4$ and $l_1 > l_3 > l_4$. The company represents its data in three relations: $(l_4, base)$, $(l_4, cargo)$ and $(l_2, value)$. The first one, $(l_4, base)$, stores the company's cars; $(l_4, cargo)$ stores the assignment of cargoes to cars; and $(l_2, value)$ tells us a cargo's value in USD. The choice of l_4 , the lowest security level, as the name-space selector for $(l_4, base)$ and $(l_4, cargo)$ reflects the fact that every user knows, or needs to know, that the company keeps a record of its cars and the cargo assignments. The selector l_2 tells us that only users with a clearance of l_1 or l_2 are sup-

16. A detailed discussion of this issue and an approach to making NULL-values precise can be found, eg, in Reiter (1984).

17. Cf Wiseman (1989).

18. Cf Denning et al (1987).

19. Cf Smith/Winslett (1992).

posed to know that the cargoes' value is also stored. Formally:

- $P_{l_1} = P_{l_2} = \{(l_2, value), (l_4, base), (l_4, cargo)\}$
- $P_{l_3} = P_{l_4} = \{(l_4, base), (l_4, cargo)\}$

At the moment, the company owns two cars. The first one is parked in front of the main entrance and is known to everybody, thus, we name it (l_4, dog) . The local part of the name, *dog*, is arbitrary. The fact that this car is shared among all is reflected in its selector l_4 . The second car is hidden in a secret garage of which only users cleared at l_1 know – we name it (l_1, cat) . Thus, the relation $(l_4, base)$ comprises two tuples: $(l_4, base)((l_4, dog))$ and $(l_4, base)((l_1, cat))$. Formally, $(l_4, base)((l_4, dog))$ is a ground atomic formula of every database's language: $A_G(\Sigma)$ and $A_G(\Sigma_{l_1}), \dots, A_G(\Sigma_{l_4})$; $(l_4, base)((l_1, cat))$ belongs, of course, to $A_G(\Sigma)$ and only to $A_G(\Sigma_{l_1})$.

Now suppose that a new car is bought, which, placed in a different secret garage, should only be available to users with a clearance of l_3 and l_1 . By chance, a user at l_3 decides to name it *cat*. Congruent with the confidentiality demand and without any naming conflicts, he can enter (l_3, cat) into $(l_4, base)$ ⁽²⁰⁾. Formally, $(l_4, base)((l_3, cat))$ is only an element of $A_G(\Sigma)$, $A_G(\Sigma_{l_1})$, and $A_G(\Sigma_{l_3})$.

In the end, there is no doubt about the number of owned cars, who shares which car and which car is kept secret from whom.

Let us now briefly list some more combinations:

- $(l_4, cargo)((l_4, dog), (l_4, food))$, an element of all languages, is a transport of a cargo in a car known to everybody; requires no aliases
- $(l_4, cargo)((l_3, cat), (l_2, liquor))$, an element of only $A_G(\Sigma)$ and $A_G(\Sigma_{l_1})$, is a transport known only to l_1 in a car known to l_1 and l_3 of a cargo known only to l_1 and l_2 ; may require aliases in D_{l_2} and D_{l_3}
- $(l_2, value)((l_4, food), (l_4, 1000))$, an element of $A_G(\Sigma)$, $A_G(\Sigma_{l_1})$ and $A_G(\Sigma_{l_2})$, is a correspondence known only to l_1 and l_2 of a public cargo and a public amount; requires no aliases
- $(l_2, value)((l_2, liquor), (l_1, 2000))$, an element of only $A_G(\Sigma)$ and $A_G(\Sigma_{l_1})$, is a correspondence known only to l_1 of a cargo known to l_1 and l_2 and an amount known only to l_1 ; may require aliases in D_{l_2}

5.2 The Bell/La Padula access control model

The work of Bell/La Padula (1975) has adapted mandatory controls used in a paper-based environment to operating systems. Their mandatory access control system, also called the Bell/La Padula model, is mainly remembered for its two access control rules, the Simple-Security-Property and the *-Property. Operating systems adhering to this model possess a number of excellent security properties – the resistance against a broad range of untrustworthy programs is among the ones most often mentioned.

The model's handling of name-spaces has received little attention. Admittedly, the authors do not deal with this issue explicitly. Yet a user can name a new object any way he wants and a confiden-

20. Since this is a unary relation we do not need aliases here.

tiality violating naming conflict will never occur. The simple reason for this is that each security level has its own local universal name space and the level's name serves as a selector for a name space. The set of selectors available to a user is determined by his clearance and comprises all security levels dominated by it.

In this example we model Bell/la Padula's operating system concepts in our database.

Let $MCP = (U, O, (S, \leq), G)$ be an instance of a security-level-based mandatory confidentiality policy. Here, U are subjects. An object $o = ((l, v), c) \in O$ has a structured name (l, v) and a content c . $v, c \in \mathcal{Q}^*$, $l \in S$ and \mathcal{Q} is the system's alphabet, eg, a subset of the ASCII set. Define the labelling function $G: U \cup O \rightarrow S$ such that $G(o) = l$.

The corresponding database is quite simple. Define $D = (\Sigma, C, I)$ and $\Sigma = (N, P, \rho)$ with $N = S \times \mathcal{Q}^*$, $P = \{(z, r), (z, eq)\}$ and $\rho((z, r)) = \rho((z, eq)) = 2$. Here we must assume that there exists a smallest element $z \in S$ with respect to the partial order on S . The relation (z, eq) , the equality, is needed to express integrity constraints. (z, r) stores the objects in the following way. An object $o = ((l, v), c)$ is represented as the tuple $(z, r)((l, v), (z, c))$. This corresponds to the container-orientated view on protection: (l, v) represents both the container's name and protection requirements and (z, c) the container's content which is also shielded by the container's protection. Formally, $(z, r)((l, v), (z, c))$ is only a ground atomic formula of the language of D_l if $l' \geq l$. To ensure the uniqueness of the containers's names, we define the first attribute of (z, r) to be the relation's primary key.

The observance of the Simple-Security-Property is already guaranteed by the definition of our database. A subject u can only query the database D_l if $G(u) \geq l$. To enforce the *-Property we can restrict the update commands. Let $\tau_{u,l} = \delta \cup \iota$ be a transaction submitted by u and applied to D_l . Then $G(u) \geq l$ and $\tau_{u,l}$ has one of the following forms:

- INSERT INTO (z, r) VALUES $((l, \langle \text{local name } v \rangle), (z, \langle \text{content } c \rangle))$
- DELETE FROM (z, r) WHERE $\langle \text{any condition} \rangle$ AND $\text{name} = (l, *)$
- UPDATE (z, r) SET $\text{content} = (z, \langle \text{content } c \rangle)$ WHERE $\langle \text{any condition} \rangle$ AND $\text{name} = (l, *)$

Note that it is these restrictions that obviate the need for aliases in this example of a database.

6 Conclusion

The theory presented here addresses the semantics and the enforcement of confidentiality demands in databases at the level of function symbols and predicate symbols. Such a confidentiality demand refers to a symbol's membership in a database's language. The theory preserves the syntax and semantics of standard open databases. There is no ambiguity with respect to the entities and relations of the real-world section of an application. And there is no ambiguity with respect to the evaluation of queries and integrity constraints – in an equijoin the unique names within the joined attribute will always compute the standard result and a foreign-key constraint will always unequivocally point to the corresponding tuples.

Acknowledgements

We would like to express our sincere thanks to the anonymous referees for their time and effort. Their constructive and valuable comments have notably helped to amend this paper.

References

- Barwise, Jon. (1991) 'An introduction to first-order logic'. Ed Jon Barwise. *Handbook of Mathematical Logic*. 7th ed. Amsterdam: North-Holland.
- Bell, David Elliott, and Leonard J. La Padula. (1975) *Secure computer system: Unified exposition and multics interpretation*. MITRE Technical Report 2997. MITRE Corp, Bedford, MA.
- Cremers, Armin B., Ulrike Griefahn and Ralf Hinze. (1994) *Deduktive Datenbanken*. Braunschweig: Vieweg.
- Das, Subrata Kumar. (1992) *Deductive Databases and Logic Programming*. Wokingham, England: Addison-Wesley.
- Denning, Dorothy E., Teresa F. Lunt, Roger R. Schell, Mark Heckman and William R. Shockley. (1987) 'A Multilevel Relational Data Model'. *1987 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press. pp 220–234.
- Davey and Priestley. *Introduction into Lattices and Order*. Cambridge University Press.
- Gajnak, George E. (1988) 'Some Results from the Entity/Relationship Multilevel Secure DBMS Project'. Ed Teresa F. Lunt. *Research Directions in Database Security*. 1st RADC Database Security Invitational Workshop 1988. New York et al: Springer-Verlag, 1992. pp 173–190.
- Garvey, Thomas D., Teresa F. Lunt, Xiaolei Qian and Mark E. Stickel. (1992) 'Toward a tool to detect and eliminate inference problems in the design of multilevel databases'. Ed Bhavani M. Thuraisingham and Carl E. Landwehr. *Database Security VI*. IFIP WG11.3 Workshop on Database Security 1992. Amsterdam: North-Holland, 1993. pp 149–167.
- Landwehr, Carl E. (1981) 'Formal Models for Computer Security'. *ACM Computing Surveys* 13.3:247–278.
- Reiter, Raymond. (1984) 'Towards a Logical Reconstruction of Relational Database Theory'. Ed Michael L. Brodie, John Mylopoulos and Joachim W. Schmidt. *On Conceptual Modeling*. New York: Springer-Verlag. pp 191–238.
- Sandhu, Ravi S., and Sushil Jajodia. (1990) 'Restricted Polyinstantiation'. Ed Bhavani Thuraisingham. *3rd RADC Database Security Workshop 1990*. Bedford, Massachusetts: Mitre, 1991. pp 13–25.
- , —. (1991) 'Honest Databases that can Keep Secrets'. *14th National Computer Security Conference*. NIST/NCSC. pp 267–282.
- , —. (1992a) 'Polyinstantiation for Cover Stories'. Ed Yves Deswarte, Gerard Eizenberg and Jean-Jaques Quisquater. *Computer Security – ESORICS 92: Second European Symposium on Research in Computer Security*. LNCS, vol 648. Berlin et al: Springer-Verlag. pp 307–328.
- , —. (1992b) 'Eliminating Polyinstantiation Securely'. *Computers & Security* 11.6:547–563.
- Smith, Kenneth, and Marianne Winslett. (1992) 'Entity Modeling in the MLS Relational Data Model'. *18th VLDB Conference*. pp 199–210.
- Spalka, Adrian, and Armin B. Cremers. (1996) 'An Axiomatic Interpretation of Confidentiality Demands in Logic-Based Relational Databases'. Ed Dino Pedresci and Carlo Zaniolo. *Logic in Databases*. LNCS vol 1154. Berlin et al: Springer-Verlag. pp 303–319.
- Thuraisingham, Bhavani M. (1991) 'A Nonmonotonic Typed Multilevel Logic for Multilevel Secure Data/Knowledge Base Management Systems'. *The Computer Security Foundations Workshop IV*. IEEE Computer Society Press. pp 127–138.
- , —. (1992) 'A Nonmonotonic Typed Multilevel Logic for Multilevel Secure Data/Knowledge Base Management Systems – II'. *The Computer Security Foundations Workshop V*. IEEE Computer Society Press. pp 135–146.
- Ullman, Jeffrey D. (1988) *Principles of database and knowledgebase systems*. Vol I and II. Computer Science Press.
- Wiseman, Simon. (1989) 'On the Problem of Security in Databases'. Ed David L. Spooner and Carl E. Landwehr. *Database Security III*. IFIP WG 11.3 Workshop on Database Security 1989. Amsterdam: North-Holland, 1990. pp 301–310.

Objected-Oriented Systems

Chair: Ravi S. Sandhu

A Principled Approach to Object Deletion and Garbage Collection in Multilevel Secure Object Bases

E. Bertino E. Ferrari

Dipartimento di Scienze dell'Informazione

Università di Milano

20135 Milano, Italy

Abstract

This paper introduces guidelines aiming at the prevention of illegal information flows due to object deletion in multilevel secure object database management systems (ODBMSs). Although a delete operation can be viewed as a kind of write operation, this does not suffice to avoid covert channels. Hence, the attention is focused on delete operation and its implications on database security. The guidelines we propose are formally stated as security principles. We also show how to design a garbage collection mechanism in a multilevel secure ODBMS. The garbage collection ensures both security and referential integrity.

1 Introduction

Object-oriented database management systems and recent object-relational database management systems (in what follows we will refer to both kind of systems as object database management systems - ODBMSs for short) continue to be an active research area for both the academic and the industrial world. Issues related to security and privacy have been investigated in the area of ODBMSs and models have been proposed for both mandatory access controls [15, 18, 23] and discretionary access controls [21]. However, much work is still needed in this area. In particular, even if some approaches, developed for relational DBMSs, can be directly applied to ODBMSs, new security problems arise that are specific to objects manipulation in ODBMSs. We believe that addressing such security issues is important given the relevance of object technology in the current and next generations of DBMSs. Moreover, security is today an important concern in many object-oriented platforms for distributed computing and application development, as witnessed by recent efforts for developing a security standard for CORBA [7]. Therefore, even though we cast our research in the framework of ODBMSs, results of our research can be applied to other object systems.

An important issue in ODBMSs is related to object deletion. Two different approaches are used by existing ODBMS to enforce object deletion; under the first approach users are allowed to explicitly delete objects; under the second approach a garbage collection mechanism is used by which an object is removed by the system when no longer reachable by other objects. Because under the latter approach no explicit delete operation is provided at application level, referential integrity is ensured. However, object deletion and garbage collection are operations that, if not properly implemented, could be exploited as covert channels, thus bypassing the access controls usually implemented by ODBMSs. An important requirement for those operations is therefore to be secure from covert channels and, at the same time, to ensure referential integrity among objects in the database. Because of the relevance of garbage collection in object systems, several algorithms have been proposed for both centralized and distributed systems [13, 17, 19]. Here, we continue our investigation in object deletion and secure garbage collection [4]. We first show how object deletion and garbage collection could be illegally exploited to perform unauthorized data accesses; then, we introduce some principles ensuring a secure delete operation. Moreover, we present a garbage collection protocol which is secure against covert channels. The main differences between the work presented in this paper and our previous work [4] can be summarized as follows. First, here we provide a formal setting to address secure object delete operations. Second, in our previous paper the copying approach to garbage collection was considered. Here, we consider a different approach, based on the mark-and-sweep technique, and show how the proposed approach is secure with respect to the formal setting. In particular, the formal setting we propose consists of a number of principles forming the basic guidelines for secure object deletion and garbage collection. These guidelines provide

a concrete *embodiment* of the general Bell-LaPadula principles [1] for the specific case of object deletion and garbage collection. We believe that such guidelines are an important step towards the development of secure object systems.

The remainder of this paper is organized as follows. Section 2 briefly recalls the basic concepts of multilevel security and outlines the object model we refer to in this paper. Section 3 describes the create operation in the framework of the object model presented in Section 2. Section 4 introduces the problem of object deletion in a secure object environment and provides rules for secure object deletion. Section 5 analyzes the mark-and-sweep garbage collection protocol with respect to the principles presented in Section 4. Finally, Section 6 concludes the paper and outlines future work.

2 Preliminary Concepts

In this section we first recall the basic concepts of multilevel security and describe the message filter model [15]. Moreover, we briefly characterize the reference object model we refer to in the paper.

2.1 The Multilevel Security Model

The system consists of a set O of objects (*passive entities*), a set S of subjects (*active entities*), and a set Lev of security levels with a partial ordering relation \leq . A level L_i is *dominated* by a level L_j if $L_i \leq L_j$. Moreover, a level L_i is *strictly dominated* by a level L_j (written $L_i < L_j$) if $L_i \leq L_j$ and $i \neq j$. We say that two levels L_i and L_j are *incomparable* (written $L_i <> L_j$) if neither $L_i \leq L_j$ nor $L_j \leq L_i$ holds. A total function \mathcal{L} , called *security classification function*, is defined from $O \cup S$ to Lev . Given an object o , function \mathcal{L} returns the security classification of o . Similarly, given a subject s , $\mathcal{L}(s)$ denotes the security classification of s .

A secure system enforces the Bell-LaPadula restrictions that can be stated as follows [1]:

1. A subject s is allowed to read an object o if and only if $\mathcal{L}(o) \leq \mathcal{L}(s)$ (no-read-up).
2. A subject s is allowed to write an object o if and only if $\mathcal{L}(s) \leq \mathcal{L}(o)$ (no-write-down).

The second property is also known as the **-property* and prevents leakage of information due to Trojan Horses. Additional details can be found in [8].

2.2 The Reference Object Model

An object database consists of a set of objects exchanging information via messages. An object consists of a unique object identifier (OID), which is fixed for the whole life of the object, and a set of attributes,

whose values represent the state of the object. The value of an attribute can be an object or a set of objects. Moreover, an object has a set of methods encapsulating the object state. Methods are used to modify the state of the object or to perform other types of computation on the object.

An object can be primitive (like an integer, or a character), or can be built from other objects (either primitive or non-primitive). We denote a non-primitive object as a triple $(oid, state, meths)$, where:

- oid is the object identifier;
- $state = (a_1 : v_1, a_2 : v_2, \dots, a_n : v_n)$,¹ where a_i is an attribute name (the names of object attributes must be distinct), and v_i is the value of attribute a_i and can be a primitive object or an OID, $1 = i, \dots, n$. The possible values that an object attribute may take are specified in the definition of the class to which the object belongs to;
- $meths$ is a set of method names.

Let o and o' be two objects. We say that o is a *high-level (low-level)* object with respect to o' , if $\mathcal{L}(o') < \mathcal{L}(o)$ ($\mathcal{L}(o) < \mathcal{L}(o')$). Similarly, let o and o' be two objects such that o stores in one of its attribute the OID of o' . We say that the OID of o' is a *high-level (low-level)* OID with respect to o if $\mathcal{L}(o) < \mathcal{L}(o')^2$ ($\mathcal{L}(o') < \mathcal{L}(o)$).

Whenever an object o has as value of one of its attributes the OID of an object o' , we say that o *references* o' .

In our model, no reference is allowed among objects at incomparable security levels. Moreover, we make the assumption, common to most proposals [15, 18, 23], that all objects are single-level and, therefore, all attributes of an object have the same security level. Multilevel objects can easily be represented in terms of single-level objects; we refer the reader to [2, 3] for a detailed discussion on this issue. [2, 3] do not address the problem of object creation; however, the creation of a multilevel object can be performed by creating a single-level object for each distinct security level. The creation of these objects can be performed

¹We make the assumption that non-primitive objects are built using the tuple constructor. Other constructors may also be used, like the set and list constructors [5]. However, the specific constructor type used is not relevant for the present discussion.

²This is possible because we allow an object to create objects at strictly higher levels (see Section 3 for more details on the create operation).

using a covert-channel free OID generation mechanism as the one illustrated in Section 3.

The methods of an object can be invoked by sending a message to the object. Upon the reception of the message, the corresponding method is executed, and a reply is returned to the object sending the message. The reply can be either an OID, a primitive object, or a special *nil* value, that denotes that no information is returned.

The invocation of a method *m* on the reception of a message can be either synchronous or asynchronous. In the former case, the sender waits for the reply value, that is, it is suspended until the invoked method terminates. In the latter case, a *nil* reply value is immediately returned to the sender which will be executed concurrently with the receiver; the sender will be able to get the reply value successively. In this paper we do not distinguish between synchronous and asynchronous method invocations. Moreover, we assume that method invocations are performed sequentially during a user session within the system, that is, a user cannot invoke parallel method executions.

The fact that messages are the only means by which objects can exchange information makes information flow in object systems have a very concrete and natural embodiment in terms of messages and their replies [15]. Thus, information flow in object systems can be controlled by mediating message exchanges among objects.

2.3 The Message Filter Model

The Bell-LaPadula model has been applied to the object model by means of the message filter [15]. Under this approach all messages exchanged among objects in the system are filtered according to the following rules:

1. If the sender of the message is at a level strictly dominating the level of the receiver, the method invoked by the message is executed by the receiver in *restricted mode*, that is, no update can be performed. More precisely, a restricted mode execution at a level *l* should be *memoryless* at level *l*. Therefore, even though the receiver can see the message, the execution of the corresponding method on the receiver should leave the state of the receiver (as well as of any other object at a level not dominated by the level of the receiver) as it was before the execution.
2. If the sender of the message is at a level strictly dominated by the level of the receiver, the method is executed by the receiver in *normal mode*, but

the returned value is *nil*. To prevent timing channels, the *nil* value is returned to the sender before actually executing the method.

The first principle ensures that an object does not write-down, whereas the second one ensures that an object does not read-up. The *message filter* is a trusted component of the object system in charge of enforcing the above principles on all message exchanges among objects. Note that, according to the reference object model, an object is allowed to reference a high-level object; this means that an object may have as value of one of its attributes a high-level OID. This possibility allows an object to send information to objects at higher levels. However, since every message is intercepted by the message filter, this possibility does not violate the overall security of the system since read-up operations will always return a *nil* response value. Moreover, an object of level *L_i* may only store the OIDs of the high-level objects whose creation has been requested by a level lower than or equal to level *L_i*. The mechanism we adopt for OIDs generation is described in the following section.

3 Create operation

From a security perspective, create is an important operation since it establishes the visibility of object OIDs across security levels. The create operation allows a subject (or an object) to create an object with a security level higher than or equal to the level of the creator. Obviously, a subject (object) cannot create objects at strictly lower levels than its security level. The create message has as arguments the list of attribute values (either primitive objects or OIDs) and the security level to be assigned to the created object. The OID of the created object is returned to the object that has requested the creation. A consequence of this approach is that objects may store, as part of their state, OIDs of objects at higher levels.

We make several assumptions about the OIDs. First, OIDs are *logical*, that is, they do not contain information about the physical location of the corresponding objects. Given an OID, a hash table is used to determine its physical location. Second, there is a separate OID generation mechanism at each level. The OIDs generated at a level *L* are for the object whose creation has been required at level *L*. Finally, we assume that the OID of each object also contains the security level assigned to the object upon its creation. This assumption does not introduce any security flaw, since the level of the object is specified as part of the create operation. Thus, the OID of an object *o* consists of three components: *L*, *L'* and *c*, where *L* is the

level of the creator of o , L' is the level assigned to o upon its creation, with $L \leq L'$, and c is an integer that uniquely identifies o at level l .

4 Secure Delete operation

We start by describing the most common approaches to object deletion and the security problems the delete operation can cause. Then, we present a set of principles ensuring the security of the delete operation for mandatory access control. Finally, we discuss implementation issues.

4.1 Delete Operation

Existing ODBMSs use different approaches with respect to the delete operation. There are two categories of systems: systems supporting explicit delete operations (like Orion [12] and Iris [11]), and systems using a garbage collection mechanism (like O2 [10] and Gemstone [16]). A garbage collector is a piece of software that deletes objects no more accessible. There is a special object, called *root*, which is always persistent. All objects that can be reached from the root by traversing (directly or indirectly) object references, are persistent. An object is removed when it can no longer be reached from the root.

If the delete operation is not properly executed, covert channels may be established.

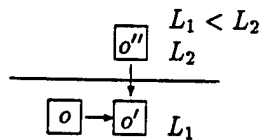


Figure 1: An object o' with references from its own level and a higher level

Example 1 Consider two objects o and o' such that $\mathcal{L}(o) = \mathcal{L}(o') = L_1$, and an object o'' such that $\mathcal{L}(o'') = L_2$, with $L_1 < L_2$. Suppose that both o and o'' reference object o' . The example is graphically illustrated in Figure 1. Suppose that the reference from o to o' is removed, because o has been deleted. If a garbage collection approach is used, object o' would not be deleted since there is still another object (i.e., o'') referencing it. Therefore two subjects at levels L_1 and L_2 , respectively could exploit this fact to establish a covert channel. The subject at level L_1 would create two objects o and o' , at its own level, such that o references o' . Then, the subject at level L_2 would create an object o'' , at its own level, such that o'' references

o' . Then, after an amount of time pre-defined by the two subjects, the subject at level L_1 would remove the reference from o to o' . If, after the reference has been removed, object o' still exists, this situation is interpreted as 1. By contrast, if object o' is removed, this situation is interpreted as 0. Note that the subject at level L_1 would simply need to check storage occupancy to determine whether o' still exists.

Exploiting the above covert channel requires collusion of two subjects at different levels. Note, however, that this is a common situation for many types of covert channels. See as an example, covert channels exploiting concurrency control mechanism in DBMS [14].

Moreover, whenever storage is deallocated because of object deletion, the problem of *dangling references* may arise. A dangling reference occurs when there is a reference to storage that has been deallocated. In systems with explicit delete operations dangling references may arise since an object can be removed even if there are references to it. In a garbage collection environment, an untrusted collector could intentionally remove an object to create a dangling reference.

A security problem is that dangling references can be used to establish covert channels, as the following example shows.

Example 2 Consider Figure 2(a). If object o' is deleted by a subject at level L_2 , a dangling reference appears in object o at level $L_1 < L_2$ (Figure 2(b)). A subject at level L_1 could infer the deletion of object o' by trying to send a write message to the object. On the basis of the result of such operation (run-time error or successful update), the subject at level L_1 gets one bit of information from a higher security level.

Thus, the deletion of objects referenced by low-level objects can be exploited by low-level subjects to infer information from high-level objects. A subject at a security level L_2 could delete a subset of the objects referenced by objects at a security level $L_1 < L_2$. Then a subject at level L_1 could try to access all high-level objects resulting in a set of unsuccessful-successful accesses. Hence, an arbitrary string of bits of reserved information could be transmitted from a higher security level. Note that, in a garbage collection environment an untrusted collector could intentionally remove the objects at level L_2 referenced by low-level objects in order to establish a covert channel.

As Examples 1 and 2 above show, there are many ways in which a delete operation can be exploited to establish a covert channel. However, it is important

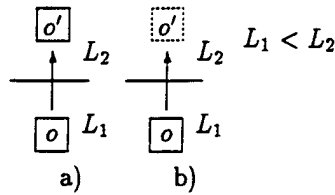


Figure 2: An object o referencing a high-level object

to note that, when an object is deleted, the only side effects on the database are a state transition of the database itself and, possibly, the generation of dangling references. Therefore, the only means to establish a covert channel exploiting object deletion are those stated by the following definition.

Definition 1 (Delete Covert Channel) A delete covert channel is a covert channel established by one of the following means: (i) exploiting dangling references; (ii) monitoring the state of the system with respect to object or memory allocation;³ or (iii) performing intentional data scavenging.⁴

In the above definition, with the term *state of the system* we refer to the state of all objects stored in the system and all the information related to the system itself, such as memory allocation and method error codes.

Moreover, delete operations can be regarded as a form of write operations: depending on the specific delete approach used, information may have to be set into the object being deleted (such as reference counts). It is therefore necessary to avoid that they can be exploited to establish a Trojan Horse. The Trojan Horse is simply established by having at a high-level some piece of code allowing or disallowing deletions of lower level objects or by plainly writing sensitive data into lower-level system information, used when removing the object (we call it *Delete Trojan Horse*). The above considerations lead to the following definition of *secure delete operation*.

Definition 2 (Secure Delete Operation) A delete operation is secure if and only if it cannot be exploited to establish a delete covert channel or a delete Trojan Horse.

³When we speak of object allocation rather than memory allocation, we mean information about whether or not memory is allocated to a given object regardless of the amount of memory it uses (e.g. the result of a query looking for the instances of a given class).

⁴Accesses to system resources such as memory pages and disk sectors no more allocated. See also *object reuse* in [6].

An important question is whether there could be other circumstances, besides the ones considered in Definition 2, leading to insecure delete operations. We believe not. Indeed, illegal information flow may arise in two cases: 1) write-down operations, which for delete operations mean that a high-level subject may cause or prevent the deletion of a low-level object, or write sensitive data into lower-level system information. This case has been identified as Delete Trojan Horse in the above definition; 2) read-up operations, which for delete operations mean that dangling references⁵ may arise, or that some low-level subjects may read high-level information about memory occupancy or de-allocated areas. All these situations have been identified as delete covert channels in Definition 1. Note that completeness of Definitions 1 and 2 is based on observing that a delete operation consists of two steps: (a) logically removing the object (and then checking references to the object); (b) physically removing the object (and thus de-allocating the storage allocated to the object). Our definitions are based on analysis of security threats that can arise in the above steps.

Even though the delete operation can be thought of as a form of write, because of the many ways the delete operation is implemented in object systems, it is important to establish some basic principles, enforcing secure delete operations, underlying any possible implementation of the delete operation. These principles are the topic of the following subsection.

4.2 Security Principles for Object Deletion

In the following we define a set of principles, referred to as *security principles*, ensuring the security of a delete operation. These principles state *what* needs to be done by the *Trusted Computing Base (TCB)* [6] to prevent illegal flows of information due to object deletion, rather than *how* it will actually be implemented. For instance, Figure 2 only shows how to exploit dangling references to establish a delete covert channel, regardless of implementation details. Indeed, the TCB can easily block these illegal flows by simply using a strategy like the one discussed in Subsection 2.3 for handling messages sent to high-level objects. According to such strategy, a message sent from a low-level object to a high-level object always returns *nil*, independently from the actual execution of the method invoked by the message.

We do not make any assumption whether deletion is implicit or explicit or whether referential integrity is enforced. In order to make our approach widely

⁵This because the delete operation just removes objects.

applicable, we do not assume a particular mandatory security model and start from the basic mandatory access control principles introduced in Subsection 2.1. Moreover, our principles do not assume any system architecture (single-subject vs kernelized).

Since delete operations can be regarded as a form of write operations, deleting a low-level object can be interpreted as a violation of the *-property. Hence, we suggest the following principle:

Principle 1 (No Delete Down) *An object o can cause the deletion of an object o' if and only if $\mathcal{L}(o) \leq \mathcal{L}(o')$.*

Note that we have used 'can cause the deletion' rather than 'can delete', because in a garbage collection environment an object can only cause the deletion of another object by updating all references to the given object, causing its deletion by the collector. In systems supporting explicit deletions, an object can cause the deletion of another object by issuing a delete command.

As we have seen in Example 1, an object o at level L could be referenced by several high-level objects and these references from high-level objects to low-level objects could be used to establish a delete covert channel. In order to prevent this type of problem, the following principle is established:

Principle 2 (No Interference from High to Low) *If an object o is referenced by high-level objects and by no object of level $L' < \mathcal{L}(o)$, a delete operation invoked on the object o from level $\mathcal{L}(o)$ cannot be prevented.*

In Example 2, dangling references from low-level objects to high-level objects are used to infer higher level data. However, OIDs referencing high-level objects are needed if write-up is allowed by the security model. Therefore, we propose the following principle:

Principle 3 (No Dangling References from Low to High because of High-Level Deletions) *A delete operation invoked from level L on an object o' of level L' , $L \leq L'$, must not be allowed if there exists at least an object o such that $\mathcal{L}(o) < L$ and o references o' .*

Note that Principle 3 does not forbid the deletion of an object o , referenced by low-level objects, if this deletion is required by an object at a level dominated by all the levels of the objects referencing o . Indeed, the dangling references arising from this deletion cannot be exploited as a covert channel trying to access the deleted object.

As stated by Definition 1, dangling references are not the only mean of establishing a delete covert channel. For instance, using an untrusted garbage collector that acts on the entire database, an object at level L could infer the deletion of an object at level L' , $L < L'$, by monitoring the system resources. Hence, system resources must be controlled according with the following principle:

Principle 4 (No Global Information) *Information about system resources at security level L can be made available to an object o if and only if $L \leq \mathcal{L}(o)$.*

It is important to note that the no read-up principle is normally intended as a restriction imposed on the operations acting on the database, whereas Principle 4 states a more general rule to avoid also leakage of information due to system information, like memory allocation.

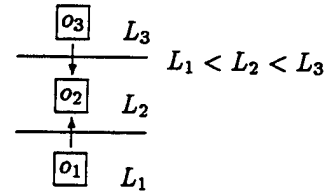


Figure 3: An object referenced from multiple levels

Example 3 *The interplay among the given principles is illustrated with the help of Figure 3. Here, an object o_2 is referenced by a high-level object o_3 and a low-level object o_1 . Suppose now that a subject at level L_2 invokes the deletion of object o_2 . Under the security principles, object o_2 is not deleted, since its deletion would violate Principle 3. Note that, even if the deletion of object o_2 is not allowed, Principle 2 is satisfied too, because object o_2 is referenced by both a high-level object (i.e., object o_3) and a low-level object (i.e., object o_1). Moreover, if a subject at level L_1 invokes a delete operation on object o_2 , this operation is allowed since the deletion of object o_2 does not violate any principle. In particular, Principle 3 is satisfied because the deletion is invoked from level L_1 . Indeed, the dangling references caused by this deletion, that is, the references from o_3 to o_2 , and from o_1 to o_2 cannot be exploited as a delete covert channel.*

The correctness of the above security principles is stated by the following proposition.

Proposition 1 *A delete operation is secure iff Principles 1- 4 are satisfied.*

Proof. We first prove the if part of the thesis. We suppose that the implication does not hold and derive a contradiction. Suppose that the delete operation is secure, and that one among Principles 1- 4 is not satisfied.

- It is trivial to prove that if Principle 1 is not satisfied the delete operation is not secure, since a delete Trojan Horse could be established.
- If Principle 2 is not satisfied, the delete operation is not secure, since a delete covert channel, like the one in Example 1, could be established.
- If Principle 3 is not satisfied, the delete operation is not secure, since a delete covert channel, like the one illustrated in Example 2, could be established.
- If Principle 4 is not satisfied, a delete covert channel could be established by monitoring the system resources at a higher security level, and hence inferring the deletion of high-level objects.

Thus, in all the above cases, a contradiction arises. We now consider the other part of the implication. We suppose that the implication does not hold and derive a contradiction. Suppose that Principles 1- 4 are satisfied and the delete operation is not secure. According to Definition 2, this is equivalent to suppose that Principles 1- 4 are satisfied and the delete operation can be exploited to establish either a delete covert channel or a delete Trojan Horse.

- Suppose that the delete operation can be exploited to establish a delete Trojan Horse. It is easy to show that this implies that the delete operation does not satisfy Principle 1, because Principle 1 is a specialization of the *-property to the delete operation context. Thus, a contradiction arises.
- Suppose that the delete operation can be exploited to establish a delete covert channel. According to Definition 1, a delete covert channel can be established only by i) exploiting dangling references, ii) monitoring the state of the system with respect to object or memory allocation, or iii) performing intentional data scavenging. We consider each of the above cases separately.

– Suppose that the delete operation can be exploited to establish a delete covert channel by means of dangling references arising from the delete operation. Downwards dangling references, as well as dangling references within the same level cannot be exploited to establish a covert channel, since they cannot be used to transfer information from a security level to lower security levels. Hence, the delete operation can be exploited to establish a covert channel only if it results in upwards dangling references. Thus, suppose that the delete operation results in a dangling reference from a level L_1 to a level L_2 , $L_1 < L_2$. This means that the delete operation removes an object from level L_2 . A delete covert channel can be established only if the deletion is required from a level L' strictly dominating level L_1 because only in this case the dangling reference can be used to transfer higher-level information, i.e. information at level L' , to a subject at level L_1 . If the delete operation is invoked from a level strictly dominating level L_2 , Principle 1 is not satisfied. If the deletion is required from a level dominated by L_2 , a violation of Principle 3 arises. Hence, in both cases, a contradiction arises.

– Suppose that the delete operation can be exploited to establish a delete covert channel by monitoring the state of the system with respect to object or memory allocation or by performing intentional data scavenging. In this case, a delete covert channel can be established only if the above operations can be used to infer higher-level information. This is possible only in two cases: 1) a subject at a given level performs the above operations at security levels strictly dominating its security level. However, in this case, Principle 4 is not satisfied, which contradicts the assumption; 2) the execution of the above operations at a given security level allows higher-level information to be inferred, that is, the result of these operations at a given security level is determined by operations at higher-levels. This is possible only if the presence or absence of an object o at a given level is conditioned by high-level objects, that is, object o is referenced by a high-level object. In this case, if o is not referenced by low-level objects and

if a garbage collection approach is used, a subject at a level strictly dominated by the level of the object referencing o could infer the existence of an object at a higher-level by invoking the deletion of o . If o is not deleted, the existence of a high-level object referencing o is inferred. Thus, consider two objects o and o' such that $\mathcal{L}(o) < \mathcal{L}(o')$, o' references o , and o is not referenced by low-level objects. Information from level $\mathcal{L}(o')$ to lower security levels can be transferred only if the deletion of object o is invoked from a level L strictly dominated by $\mathcal{L}(o')$, because only in this case the deletion or non deletion of object o can be used at level L to infer higher-level information, that is, information at level $\mathcal{L}(o')$. If the level from which the deletion is invoked strictly dominates level $\mathcal{L}(o)$, Principle 1 is not satisfied, which contradicts the assumption. If the deletion is invoked from a level L strictly dominated by $\mathcal{L}(o)$, then a delete covert channel can be established only if information on the allocation of object o is accessible at level L . However, in this case Principle 4 is not satisfied, which contradicts the assumption. Finally, if the deletion is required from level $\mathcal{L}(o)$, information from level $\mathcal{L}(o')$ can be inferred only if the deletion is not allowed. In this case, Principle 2 is not satisfied, which contradicts the assumption.

4.3 Implementation Issues

As stated in the above discussion, the four principles guarantee the security of the delete operation. The description of a detailed implementation for the delete operation is outside the scope of this paper. Nevertheless, it is possible to make some implementation-independent considerations to help in securely designing and implementing such operation.

- The delete operation can be considered a special case of write operation. Therefore a *TCB* enforcing the $*$ -property verifies Principle 1 (for more details about garbage collection see Section 5).
- The delete operation must neither update the state of an object nor read it, but it must physically remove an object. An important requirement for a system to be secure is that the basic storage elements (e.g. disk sectors, memory pages, etc.) be cleared prior to their assignment to an object so that no intentional or unintentional data scavenging takes place. The storage elements can

be cleared when deallocated, that is, when an object is deleted. The security principles are defined disregarding implementation details. Hence we require the physical deletion to be performed by the *TCB*: when a delete operation is invoked on an object, the *TCB* calls a trusted procedure to perform the deletion.

- There are two possible approaches to enforce Principle 3:
 1. Upwards dangling references are masked by concatenating the OIDs with the security level where the object is allocated [18] and making such dangling references *ineffective* [4]. That is, an object trying to access a high-level object is returned a default reply value even if the target object has been previously deleted. This can be performed by the *TCB* that determines the security level from the OID and can, therefore, recognize a high-level OID. This approach requires that the OID contains the security level of the object (cfr. Section 3).
 2. The deletion of an object like o' in Figure 2 is prevented by the *TCB*.⁶

In both cases Principle 3 is verified. In particular, the first solution makes the upwards dangling references ineffective. Hence the principle is merely satisfied since no low-to-high dangling reference can compromise security. Note that this strategy can also be applied to incomparable security levels. Moreover, according to the first of the above approaches, object o_2 in Figure 3 can be deleted by a subject at level L_2 and Principle 3 is satisfied because the dangling reference from object o_1 to object o_2 , arising from the deletion of o_2 , is masked by the *TCB*. The first approach can also be adopted for the create operation. An object o requesting the creation *immediately* receives the new OID and the creation itself is executed asynchronously: errors possibly occurred during the creation do not prevent object o from immediately receiving the new OID.

Note that, if the second approach is chosen, the situation shown in Figure 4 seems to generate a security violation. Suppose that object o_2 in Figure 4(a) is being deleted by a subject at level L_2 . If a subject at level L_1 removes the OID referencing o_2 and the OID in object o_3 is not updated

⁶ Auxiliary information about low-level OIDs should be stored by the *TCB* for this purpose.

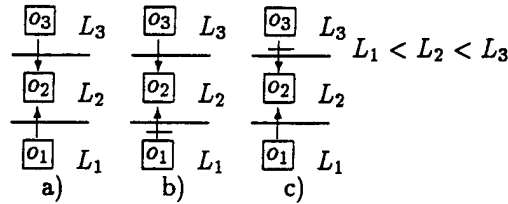


Figure 4: A suspicious delete covert channel

(Figure 4(b)), the deletion is executed (Principle 2), whereas if a subject at level L_3 removes the OID referencing o_2 and the OID in object o_1 is not updated (Figure 4(c)), the deletion is denied (Principle 3). Hence, a subject at level L_2 can infer a bit of information from the result of the deletion (denied or executed). In particular, even if the reply value of the delete operation is the same in both cases, the subject at level L_2 can infer the deletion of object o_2 by monitoring the memory allocation at level L_2 . Nevertheless, given a secure system, the example stated above is *not* a delete covert channel. Indeed:

1. In the example stated above, three subjects must cooperate. The two subjects at levels L_3 and L_1 must know not only the way the covert channel can be established, but they must also know all the data to be transmitted.
2. The subject at level L_1 cannot know the data to be transmitted to the subject at level L_2 . Such data cannot be legally transmitted from level L_3 to level L_1 via the system resources – no write-down and no read-up principles are enforced by the TCB. Moreover, a security violation cannot occur – the system is secure, in particular the security principles are enforced. Hence, if the data can be transmitted from level L_3 to level L_1 , the system is not secure against the hypothesis.
3. The subject at level L_3 is forced to *transmit* to the subject at level L_1 what the subject wants to illegally transmit to the subject at level L_2 . Hence, if the example shown in Figure 4 were a delete covert channel then each information flow would be (potentially) illegal.

A TCB satisfying the requirements stated above can be designed on the basis of the message filter ap-

proach described in Subsection 2.3. Finally, note that referential integrity is not preserved by the security principles, because they deal only with security. In particular, because of Principle 2, dangling references can arise which, however, cannot be exploited as delete covert channels.

5 Secure Garbage Collection

Our aim is to achieve referential integrity in multilevel databases by means of garbage collection. Garbage collection deals with reclaiming storage that was once used but is no longer needed. The collector is invoked periodically or when a memory overflow arises. A serious drawback with conventional garbage collection mechanisms is that the garbage collector would have to access objects at various security levels. The garbage collector would therefore have to be trusted. We describe here a different approach, based on the *mark-and-sweep* technique; under this approach the collector is structured so that the trusted part is minimized. We analyze this approach with respect to the four principles for secure delete operations and we show that the garbage collector satisfies the four principles. It, therefore, implements a secure delete operation, because of the results stated by Proposition 1.

A mark-and-sweep collector follows pointers in the heap marking any object that is reached (*marking phase*), then it collects all the non-marked objects (*sweeping phase*) scanning the heap sequentially. The marking phase starts from the *root objects* (objects containing information always needed). The root objects are the “entry points” for a security level.

One way to implement a multilevel trusted collector is to employ a TCB [22] which controls the behavior of single-level untrusted collectors and enforces the Bell-LaPadula principles. Therefore, we require root objects and a marking collector MC_L for each security level L . The marking collector MC_L is an object at level L in the database, which is activated and controlled by the TCB. The marking collector MC_L executes the marking phase for level L , while all sweeping

phases are performed by the *TCB* to avoid data scavenging. MC_L does not mark an object o at level L or higher if and only if all references to o from other objects at level L have been removed (that is, the object is *non-locally reachable*). Garbage collection is managed according to the stop-the-world approach: activities are suspended, garbage is collected and then activities are restarted.

Since marking collectors are untrusted objects, each marking collector can only read objects or system information at its security level or at lower levels. In the following we show how to prevent the marking collectors from being exploited as storage covert channels. Storage covert channels are illegal channels established via the exploitation of the dynamic allocation of memory or via data scavenging. For example, a high-level subject could establish such a covert channel by saturating the memory, to prevent the normal computation of a low-level subject, which in turn could infer high-level information. To overcome this drawback, we adopt the following solution. System memory (volatile and non-volatile) is divided into a number of partitions of fixed size, one for each security level. Subjects at level L can allocate memory only from the partition assigned to L and the creation of a high-level object is performed at the level requested for the new object. This allocation scheme prevents storage covert channels from being established.

The marking collector MC_L executes a write operation in order to mark an object, hence it is only able to mark objects at level L or higher. The marking collector MC_L cannot be aware of references from objects at security levels higher than L because of the no read-up restriction. Therefore, dangling references could arise at security levels higher than L after the garbage collection is completed. The approach we propose to avoid dangling references is based on copying operations. Under this approach, an object o , non-locally reachable at its security level, is copied at higher security levels as needed. This mechanism does not need to be trusted; therefore it can be implemented by the marking collectors. The marking collector MC_L ⁷ builds a table called *Copy Table* to store pairs of related OIDs of the form (*old-oid*, *new-oid*), where *old-oid* is the OID of a low-level non-marked object while *new-oid* is the OID of its copy created at level L by the marking collector MC_L . The *Copy Table* at level L is read by the marking collectors at levels higher than L to avoid redundant copies. It is sufficient to create a copy of an object o non-locally reachable, at the lowest levels where o is needed and

update all high-level objects referencing o . When dealing with incomparable levels, a copy is generated for each level as needed.

The marking collectors are activated by visiting the security lattice on the basis of a sequence (L_1, \dots, L_n) called *visit-sequence*, where L_1 is the lowest level, L_n the highest and for each L_i, L_j , $1 < i \leq n, 1 \leq j < i$, $L_j < L_i$ or $L_j \nless L_i$. The visit-sequence is a static list associated with a given database. When level L is visited, the marking collector MC_L is activated and after its termination the next security level in the visit-sequence is visited. The sweeping phases must be postponed till the end of the marking phases, otherwise an object could be removed before being copied: when the last collector completes its execution, the sweeping phase is performed for all security levels.

Figure 5 shows an example of this approach: object 1 at level L_1 is not marked by the marking collector MC_{L_1} ; hence it is copied at level L_2 by the marking collector MC_{L_2} that adds the pair (*oid*(1), *oid*(1')) to the *Copy Table* at level L_2 . The *OID* stored in object 4 at level L_3 and referencing object 1 is updated with the *OID* of the copy 1' generated at level L_2 . This update is performed by the marking collector MC_{L_3} by reading the *Copy Table* at level L_2 .

This approach satisfies the security principles previously stated. Suppose that the marking collectors correctly execute the marking phase. We have that:

- Principle 1 is satisfied. The marking collectors cannot perform explicit deletions. Moreover, they are objects under the control of the *TCB*; hence they cannot violate the *-property by causing a low-level object to be deleted.
- Principle 2 is satisfied. A non-marked low-level object is copied at higher security levels, if needed, then it is deleted by the *TCB*. By contrast, if an object is locally reachable, it is marked and cannot be deleted.
- Principle 3 is satisfied. No low-to-high dangling reference appears if the marking collectors execute correctly the marking phase.
- Principle 4 is satisfied. The rule stated by this security principle is enforced by the *TCB*; hence we can assume that the principle is satisfied.

Even if a marking collector incorrectly executes the marking phase, no security violation arises. The marking collector MC_L cannot read information at higher or incomparable levels; hence an incorrect marking phase can only generate dangling references from objects at level L . The only dangling references that

⁷Except for the lowest level in the security lattice.

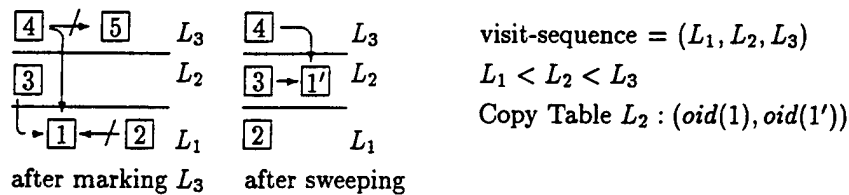


Figure 5: Achieving referential integrity with garbage collection

could be exploited to establish a delete covert channel are those referencing high-level objects that have been deleted during the sweeping phases; these dangling references can be made ineffective by the *TCB* (cfr. Subsection 4.3, solution 1). Moreover, the system memory is divided into partitions of fixed size which is always the same. Therefore, it is not possible to establish covert channels due to memory saturation, overflows and so on. Finally, timing channels due to the execution of marking collectors can be avoided by properly controlling their execution time. For instance, the execution time of each marking collector can be forced to be longer than a pre-defined lower bound, which can be the same for each security level. Since the upper bound for the bandwidth of timing channels is 100 bit per second (bps) [9], the lower bound can be assigned a value that does not cause performance penalty.

It can be shown by similar arguments, that the copying garbage collection mechanism proposed in [4] implements a secure delete operation. It moreover ensures referential integrity as the mark-and-sweep techniques described here, by copying at higher levels objects which are no longer reachable by lower levels.

6 Concluding Remarks

We have analyzed issues related to object deletion in multilevel secure ODBMSs and we have stated four security principles ensuring a secure delete operation. These four principles are rules that should be observed in designing and implementing garbage collection mechanisms and mechanisms for data manipulation in object systems. The security principles do not assume a particular security architecture, nor they define rules for a specific implementation. We have shown how a multilevel garbage collector algorithm, based on the mark-and-sweep technique, can be analyzed with respect to the above principles.

We believe that much work is still needed in this area. An important question concerns the computational overhead associated with each principle. Because the principles are abstract and are independent on any specific implementation, it is difficult to assess

the overhead. In general, we can say that the overhead is proportional to the number of references that each object has and how such references are distributed across levels. Such considerations are confirmed by some experimental evaluations performed on an implementation of the copying garbage collector. The experiments have shown that the performance heavily depends on the number of copying operations of objects from lower levels to higher levels. The number of copying operations in turn depends on the number of references from higher-level objects to lower-level objects. Another factor impacting the performance is the structure of the lattice of security levels. If the number of incomparable levels is high, the performance is good, because the first phase of the collector can be activated in parallel for all the incomparable levels. By contrast, the performance is not optimal when all levels are totally ordered. It is easy to see that these experimental considerations apply also to the mark-and-sweep garbage collector presented in this paper.

Another important topic we plan to investigate is the analysis of the rules stated by the security principles in the framework of relational [20] and deductive database systems, and their extension to cope also with object creation. Finally we plan to investigate the definition of formal strategies to state covert channels and their implications on implementation.

References

- [1] Bell D., LaPadula L. "Secure Computer Systems: Unified Exposition and Multics Interpretation". Technical Report ESD-TR-75-306, MTR-2997, MITRE, Bedford, Massachusetts, 1975.
- [2] Bertino E, Jajodia S., "Modeling Multilevel Entities Using Single-Level Objects". In *Proc. 3rd Int'l. Conf. on Deductive and Object-Oriented Databases*, LNCS 760, pp. 415-428, December 1993.
- [3] Bertino E., Ferrari E., Samarati P. "A Multilevel Entity Model and its Mapping onto a Single-Level

- Object Model". *Theory and Practice of Object Systems*, to appear.
- [4] Bertino E., Mancini L. V., Jajodia S. "Collecting Garbage in Multilevel Secure Object Stores". In *Proc. IEEE Symp. on Research in Security and Privacy*, Oakland, CA, May 1994.
 - [5] Bertino E., Martino L. "Object-Oriented Database Systems - Concepts and Architectures", Addison-Wesley, 1993.
 - [6] Chokhani S. "Trusted Products Evaluation" *Communications of the ACM*, vol. 35, no.7, July 1992, pp. 66-76.
 - [7] "CORBA Security Draft". Draft 0.2, Document Number 95-9-1, September 1995, OMG.
 - [8] Denning D. E. "Cryptography and Data Security", Addison-Wesley, 1982.
 - [9] Department of Defense. "Trusted Computer Systems Evaluation Criteria", DOD 5200.28-STD, December 85.
 - [10] Deux O. et al. "The Story of O_2 ". *IEEE Transactions on Knowledge and Data Engineering*, vol.2, no. 1, 1990, pp. 91-108.
 - [11] Fishman D. et al. "Overview of the Iris DBMS". *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, 1989, pp. 219-250.
 - [12] Kim W. et al. "Architecture of the ORION Next-Generation Database System". *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, 1990, pp. 109-124.
 - [13] Kolodner E., Liskov B., Weihl W. "Atomic Garbage Collection: Managing a Stable Heap". In *Proc. ACM-SIGMOD International Conference on Management of Data*, Boston, Oregon, 1989.
 - [14] Jajodia S., Atluri V. "Alternative Correctness Criteria for Concurrent Executions of Transactions in Multilevel Secure Database Systems". In *Proc. of the IEEE Symp. on Research in Security and Privacy*, Oakland, CA, May 1992.
 - [15] Jajodia S., Kogan B. "Integrating an Object-Oriented Data Model with Multilevel Security". In *Proc. of the IEEE Symp. on Research in Security and Privacy*, Oakland, CA, May 1990.
 - [16] Maier D. et al. "Development of an Object-Oriented DBMS". In *Proc. of the 1st OOPSLA Conference*, Portland, Oregon, October 1986.
 - [17] Mancini L. V., Shrivastava S. K. "Fault-Tolerant Reference Counting for Garbage Collection in Distributed Systems". *The Computer Journal*, vol. 34, no. 6, 1991.
 - [18] Millen J. K., Lunt T. F. "Security for Object-Oriented Database Systems". In *Proc. of the IEEE Symp. on Research in Security and Privacy*, Oakland, CA, May 1992.
 - [19] Moss J. E. "Working with Persistent Objects: to Swizzle or Not to Swizzle". *IEEE Transactions on Software Engineering*, vol. 18, no. 8, 1992.
 - [20] Qian, X. "Inference Channel-Free Integrity Constraints in Multilevel Relational Databases". In *Proc. IEEE Symp. on Research in Security and Privacy*, Oakland, CA, May 1994.
 - [21] Rabitti F., Bertino E., Kim W., Woelk D. "A Model of Authorization for Object-Oriented and Semantic Database Systems". *ACM Transactions on Database Systems*, vol. 16, no. 1, March 1991.
 - [22] Shockley W. R., Schell R. R. "TCB Subsets for Incremental Evaluation". In *Proc. of the 2nd AIAA Conference on Computer Security*, December 1987.
 - [23] Thuraisingham M.B. "Mandatory Security in Object-Oriented Database Systems". In *Proc. of the Int'l Conference on Object-Oriented Programming Systems, Languages, and Applications*, New Orleans (Louisiana), October 1989.

Compile-time Flow analysis of Transactions and Methods in Object-Oriented Databases

Masha Gendler-Fishman and Ehud Gudes
Department of Mathematics and Computer Science
Ben-Gurion University, Beer-Sheva, Israel.
e-mail: masha,ehud@cs.bgu.ac.il

Abstract

Methods are an important characteristics of Object-oriented databases, Previous models for Discretionary access-control in OO databases have considered policies for Methods and Inheritance. However, discretionary authorization models do not provide the high assurance required in systems where Information flow is considered a problem. Mandatory models can solve the problem but usually they are too rigid for commercial applications. Therefore discretionary, information-flow control models are needed, especially when transactions containing general methods invocations are considered.

This paper first reviews existing security models for object-oriented databases with and without information-flow control. These models rely on the run-time checks of every message transferred in the system. This paper uses a **compile-time** approach and presents algorithms for flow control which are applied at Rule-administration and Compile times, thus saving considerable run-time overhead. Special emphasis is put on the Flow-analysis of Methods and the transactions invoking them.

Keywords: Information Flow, Discretionary, Object-oriented, Methods

1 Introduction

Security is an important topic for Databases in general and for Object-oriented databases (OODB) in particular [Kim90, Kemp94]. In general, authorization mechanisms provided by commercial DBMS are *discretionary*, that is, the grant of authorizations on an object to other subjects is at the discretion of the object administrator.

The main drawback of discretionary access control is that it does not provide a real assurance on the satisfaction of the protection requirements, since discretionary policies do not impose any restriction on the usage of information by a subject who has obtained it legally. For example, a subject who is able to read data can pass it to other subjects not authorized to read it. This weakness makes discretionary policies vulnerable to attacks from "Trojan horses" embedded in programs.¹ Access control in *mandatory* protection systems is based on the "no read-up" and "no write-down" principles [Cast95]. Satisfaction of these principles prevents information stored in high-level objects to flow to lower level objects. The main drawback of mandatory policies is their rigidity which makes them unsuitable for many commercial environments.

There is the need of access control mechanism able to provide the flexibility of discretionary access control, and at the same time, the high assurance of mandatory access control.

¹the term "Trojan horse" is used here to refer to any illegal leakage of information, not necessarily a destructive one...

A first attempt to do it in the context of OODBs was made by Samarati et al [Samar97]. The main problem with the model in [Samar97] is that all the checks are done at *Run-time* which increases considerably the overhead in the system. Many DBMSs rely on protection which is checked at compile time! For example, Query modification in Ingres [Stone76] or View-based mechanisms in System R [Griff76] in Relational systems, or the model suggested by Fernandez et al [Fern94] for Object-oriented databases. In this paper we investigate the problem of insuring safe information flow for Object-Oriented databases by performing the checks at *Compile time* or at *Rule-definition time*, thus saving considerable overhead at run-time. A very important assumption of the present paper is that the run-time of the DBMS can be trusted. That is, if one composes its transactions only from Queries and Methods which were compiled under the control of the DBMS, one can trust their object-code and the Run-time system which executes them. A similar assumption is made in View-based systems where views are kept after they are compiled and optimized [Griff76].

In a recent paper by the same authors [GenGud97] we presented a simple Transactions model and algorithms to check for information-flow at compile-time for this transaction model. The limitations of the transaction model in [GenGud97] was that only the basic READ/WRITE methods were allowed, and no general methods. In this paper we extend the previous transactions model by allowing transactions to invoke any method (with or without parameters) and these methods may further invoke other methods. We put very few restrictions on the type of programming language and constructs used within methods. Using program-flow analysis techniques [Muchnick81], we are able to analyze the methods at compile-time when they are entered into the system, and complete the analysis at the time the transactions is compiled. Therefore, this process is very efficient since the compile-time analysis for methods is done only once (provided the method was not changed).

It is important to note that our algorithms provide an *upper-bound* for the information-flow problem. Since program-flow analysis methods cannot know the actual Run-time control-flow, they must consider all branches of an If, WHILE or CASE statements. Thus, when our algorithm reports on safe information-flow the safeness is assured, but when it reports on an unsafe information flow, it actually reports on only a potential unsafe information flow. If one is very concerned about "false" alarms, one can employ a Run-time method in these cases, such as the one in [Samar97]. Another problem in [Samar97] is that within a Method or Transaction all the information associated with a Read query is added into the overall run-time flow, regardless of whether there is an actual flow (between program statements) of this information into the objects which the Method writes. This problem is overcome in our model, because flows within program statements are analyzed within every method.

As this paper relies heavily on three previous papers [Samar97], [Fern94], and [GenGud97], these papers are first reviewed briefly in Section 2 and the definition of safe information flow is given. The generalized Transactions and Methods model is defined in Section 3 and the overall approach is explained. In Section 4 we present our compile-time analysis of Methods, and in Section 5 we present the Transactions analysis algorithm. Examples are given in both sections. Section 6 is the Summary.

2 Background

2.1 Fernandez et. al

The first model [Fern94], assumes a simple discretionary Rules-based authorization. The model deals mainly with the impact of inheritance on security and enforces several inheritance-based policies. In following papers, policies were proposed for negative authorization, content-

dependent restrictions, and for resolving conflicts between several implied authorizations (see [Larr90]). Another paper extended the basic model to include treatment of general methods [GalOz93] (As an example to the inheritance policies, consider the database in Figure 1. A rule giving a user Read access to all attributes of Student, implies also a Read access to Foreign Student's Social security number (SSN), but not to his/her Visa...)

Because of space problems we will not review this paper here. The most relevant point is the discussion of an Access Validation algorithm. The validation algorithm is applied at *Compile-time* in that it works after the Query translator and its output is entered to the Optimizer and run-time system. The Access-validation algorithm accepts two major inputs:

- The original query after translation in form of a tree. This query is further extended using the inheritance hierarchy to something called *Authorization Tree (AT_yes)*. (the AT_yes will be redefined in the next section, therefore we do not detail its structure here). Initially, all the AT_yes's nodes are authorized. After the validation algorithm, the AT_yes contains only the nodes and the attributes to which access is allowed (see an example in Section 3.1).
- The rules which are relevant to this query are extracted from a tree called the *Security Graph* which is an extension of the AT_yes upwards and downwards to include all relevant rules.

The algorithm scans in parallel the query nodes and security graph nodes, applies the inheritance policies mentioned above and produces the final AT_yes which defines the allowed access.

2.2 Samarati et. al

The second model by Samarati et al. [Samar97] describes a run-time architecture (Message filter) for checking for information flow. Again, for reasons of space we cannot describe the model in detail. The most important concepts are:

- **Transaction.** A transaction is the set of methods invocations caused by a user sending a message. The first message invokes a method which invokes other methods by sending messages to it and waiting for replies. The invoking method may in turn wait for the reply (synchronized) or deferred its waiting. A user executing a transaction is called the *Transaction initiator*.
- **Access lists.** There are several access lists associated with each object including RACL(o) - the list of users which can read from object o, WACL(o) - the list of users which can write into object o.
- **Information flow.** There exists a flow between O_i and O_j in a transaction if and only if a write or create method is executed on O_j , and that method had received information (via forward or backward transmission) on O_i . When a method *A* sends a message to another method *B*, then all the information which flowed into *A* is assumed to flow into *B*. Similarly, if a method *A* receives a reply from *B*, the information that flows into *B* is assumed to flow into *A*.
- **Safe Information flow.** Information flow is safe only if there is information flow from O_i to O_j and all users which can read O_j can also read O_i , i.e. $RACL(O_j)$ is contained in $RACL(O_i)$.

To enforce only safe information flows, [Samar97] suggests the construction of a *Message Filter* component which intercepts each and every message in the system. Using

this information it is possible to enforce safe information flow and **disallow** transferring of information which may cause an unsafe flow (i.e an empty reply is returned in that case...)

Although the above algorithm is very general and works for various types of methods and executions, it requires the check and filtering of every message in the system. This is a considerable overhead! In the next sub-section, we discuss a simpler model, of Read/Write methods only, but on which a compile-time algorithm based on [Fern94] is used.

2.3 Gendler & Gudes Model

The model by Gendler & Gudes [GenGud97] provides compile-time checking for information-flow which is based on the AT_yes idea of [Fern94]. The following concepts are used:

- **Object Model and Authorization Model.** Both models are similar to the ones in [Fern94]
- **Access Lists.** In [Fern94] the main administration structure was the *authorization rule* placed at special class/node in the object-hierarchy tree. For purposes of flow control we need to define also for each attribute a list of all users authorized to read it. We maintain the structure called *read access list* (RACL) containing the list of users who have read privileges to the attribute. The RACL of-course can be obtained using the inheritance policies mentioned above:

$$\begin{aligned} \text{RACL}(O.\text{Attr}) = \{u : (\exists O' | O \preceq O' \text{ and } \exists \text{ rule } (u, R, O'.\text{Attr})) \\ \wedge (\nexists O'' | O \preceq O'' \preceq O' \text{ and } \exists \text{ rule } (u, -R, O''.\text{Attr}))\} \end{aligned}$$

i.e. this list contains the users that the authorization rules permit them a read access to the attribute, either explicitly or via the inheritance policies specified above.

- **Authorization Tree.** Each query of the type above is validated against the initiator (U) authorization rules using the model and algorithm presented in [Fern94]. The result of such validation is the set of objects (classes) and their attributes which is authorized for this query. Basically, this set is a sub-tree of the query graph rooted at $O.\text{Attr}$ and is called *authorization tree*, denoted $\text{AT_yes}(u, A, O.\text{Attr})$. In the sequel we will usually use the authorization-trees for Read access, and therefore denote them as: $\text{AT_yes}(u, O.\text{Attr})$.
- **User Access Tree (UAT).** The set of attributes in the "entire database" that user u is allowed to access for reading is called *user access tree*.

$$\text{UAT}(u) = \{(O.\text{Attr}) : u \in \text{RACL}(O.\text{Attr})\}$$

The above UAT is computed from the data structure *RACL*, but, obviously, it is also true that

$$\text{UAT} = \cup_{i,j} \text{AT_yes}(O_i.\text{Attr}_j)$$

- **Common User Access Tree (CUAT).** We introduce a new measure for every attribute A_j : the intersection of UATs of all users who are permitted to read it. This intersection expresses the set of all attributes which is allowed to be read by all users who are allowed to read attribute A_j .

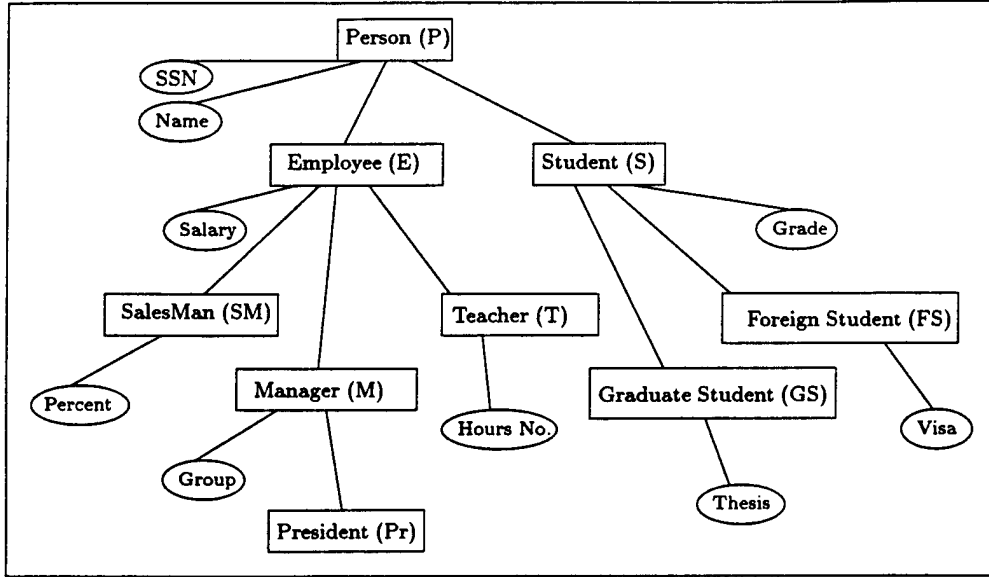


Figure 1: University database

$$\text{CUAT}(O.\text{Attr}) = \bigcap_{\forall u \in \text{RACL}(O.\text{Attr})} \text{UAT}(u)$$

As will be shown below, the CUAT is a critical component in computing safe information flow. Detailed algorithms for its computation were given in [GenGud97] .

- **Safe Information Flow.** We are now ready to define the criteria for safe information flow. Intuitively, we know that every read query after validation can only read the objects and attributes contained in that's query authorization tree. Therefore, the union of these trees expresses all the information to which this transaction has read access. We must make sure that the users who have access to the object into which this transaction writes, are allowed to access all the information that has flowed into the transaction upto the Write query.

Theorem 1 (Safe Information Flow) The information flow to the attribute $O_k.\text{Attr}_j$ caused by the write access $\text{write}(O_k.\text{Attr}_j, v)$ in a transaction, is safe, if and only if, the Common Users Access Tree of the attribute $O_k.\text{Attr}_j$ contains the union of the authorization trees of all the previous read queries.

$$\bigcup_{i=1}^{j-1} \text{AT_yes}(i) \subseteq \text{CUAT}(O_k.\text{Attr}_j) \iff \text{the information flow to } O_k.\text{Attr}_j \text{ is safe.}$$

Proof see [GenGud97]. Intuitively, each object read by the transaction and potentially written into $O_k.\text{Attr}_j$, must be contained in the set of objects allowed Read access by all users who can read object $O_k.\text{Attr}_j$.

3 The generalized Transactions & Methods Model

The transactions model presented in this section, is a generalization of the model in [GenGud97] discussed in above, in that a transaction may now call methods and pass to them parameters. Such a method may further call other methods and in return may return a value or an

object to the calling method or transaction. Both the transaction and the method it calls may issue Reads and Writes to the database, therefore, information flow may occur within methods and it must be checked for. Formally, we define:

Transaction. A transaction consists of a sequence of Read or Write queries, or Method calls, where:

Read query. $val = read(O.Attr)$ where O is a database object/class, $Attr$ is an attribute and val is the variable that receives the result.

Write query. $write(O.Attr, val)$ where O and $Attr$ are as before and val is the value (or variable) to be written into the object attribute.

Method with return value. $val = meth(p_1, \dots, p_n)$ where $meth$ is the name of a called method, $p_1 \dots p_n$ are the method's parameters (in a form of single instances or AT_yes hierarchical trees) and val is the variable that receives the return-value.

We also assume without loss of generality that a transaction does not contain any control-flow statements (they can be embedded within a method) and therefore contain only calls to the methods of the above type. As explained in the introduction, the analysis of transactions is done in two phases:

1. The Method's compile-time phase
2. The Transaction's compile-time phase.

Each method is analyzed separately at the time it is compiled and stored in the database, and the analysis results are stored in specialized security-related tables. The transaction analysis phase uses the information in these tables at the time the transaction is compiled.

3.1 Analysis of methods

In this section we discuss the compile-time analysis of a single method. This analysis is done once when the method's code is inserted into the database, and the results are stored in several tables associated with the method. The results of this analysis are composed of three parts:

1. The information flow to objects accessed for writing from within the current method.
2. The information flow to the return value of the method (if one exists).
3. The forward information flow via the parameters to methods which are called from within the current method.

These results are stored in special database tables and are used in the next stages of the analysis. Since the actual parameters to the method are not known, we use "virtual" symbols in the tables. At transaction analysis time the real information flow is substituted for these virtual symbols.

The method analysis itself uses program-flow analysis techniques which are common in Compiler and program Optimization ([Aho86, Muchnick81, Denn86]). We assume that we have a parser that can generate a syntax tree of the method, and the flow-analysis phase operates on this tree. We assume that the method is written in a programming language like Pascal or C (C++) with some restrictions. We assume that all variables in the methods are strongly-typed, i.e. pointers and memory management operators are either strongly typed too or forbidden. Another limitation is that there are no static or global variables, i.e. all information between methods is passed via the parameters, and the `goto` statement is also forbidden. We also use the following notation:

1. $FLOW(a)$, where a is an OODB object or an attribute of an object or a local variable of method - a list b of OODB objects, local variables and virtual symbols, such that there is a potential information flow to a , $b \rightsquigarrow a$.
2. IN stores the potential flow to the current block of statements ([Muchnick81]).²

3.2 Assignment statement

Let us begin with the simple case of assignment statement:

$$S \rightarrow id = E$$

Assignment is the simplest way for information flow. All information from expression E flows into variable id . Note that syntax analysis of the expression E is required to find all variables (or functions) participating in the expression. We use the notation $a \in E$ to specify the variables involved in the expression E .

$$FLOW(id) = \bigcup_{a \in E} \{a\} \cup FLOW(a) \cup IN \quad (3.1)$$

We must include IN, if this statement is part of an IF or a LOOP block as explained below.

3.3 Block of statements

We define the relation *before* between two statements as follows: S_1 *before* $S_2 \Rightarrow$ if S_1 leads to an information flow $FLOW_{S_1}$ to variable x and S_2 leads to an information flow $x \rightsquigarrow y$ such that: $FLOW_{S_1} \rightsquigarrow y$. Formally, S_1 *before* $S_2 \Rightarrow$ if $\exists FLOW(x)$ updated in S_1 - $FLOW_{S_1}$ and \exists flow $x \rightsquigarrow y$ in S_2 , such that $FLOW_{S_1} \rightsquigarrow y$ exists, and statement S_1 is executed before S_2 . Thus, the statement S_1 *must* be analyzed before S_2 . Consider the following statements block:

$$\begin{aligned} S &\rightarrow S_1; S_2; \dots S_n \\ S_1 &\text{ before } S_2 \text{ before } \dots \text{ before } S_n \end{aligned} \quad (3.2)$$

We say that inside a block of statements there are *sequential flows*. The analysis of a block of statements must therefore pass sequentially thorough all the statements in the block. We shall see in following subsections another order of flows for loop statements.

3.4 Read/Write queries

A method may contain Read queries and Write queries. Such queries will cause information flow as follows. Assume a Read query: $a = read(O.Attr)$ then

$$FLOW(a) = O.Attr \cup FLOW(O.Attr) \cup IN \quad (3.3)$$

We say that as a consequence of a read query the information flows to a from $O.Attr$ as well as from all objects and variables that the information flowed from them to $O.Attr$ before the Read. For a Write query: $write(O.Attr, a)$ then

$$FLOW(O.Attr) = a \cup FLOW(a) \cup IN \quad (3.4)$$

² we will use in the rest of the paper the term "information flow", although it should be clear that we mean essentially only "potential information flow"

The information flows to $O.Attr$ from a , as well as from all objects and variables that the information flowed from them to a and to the Write statement.

Now, for every Write query we must record the flow that was caused by the Write in a table entry, since later on we will analyze whether that flow was safe or not. Each compiled method will have its own WRITES table containing one entry for every Write query in that method. An entry in the WRITES table has the format: $(Obj, WrInFlow)$, where Obj is OODB object accessed for writing and $WrInFlow$ is set of OODB objects and virtual symbols that is contained in $FLOW(a)$ (Note that local variables are not inserted into this table).

3.5 If statement

Consider the following example:

```
if (a > 1)
  b = 1;
else  b = 2;
```

There is no direct information flow from a to b in this example. However, it is possible after the execution of the code to draw a conclusion from the value of b about the value of a . So there is a potential information flow from the **if** condition to both the **then** and the **else** branches. (see [Denn86] for an extensive discussion of this example). Formally,

$$S \rightarrow \text{if}(E) S_1; \text{ else } S_2;$$

$$IN(S_1) = IN(S_2) = \bigcup_{a \in E} \{\{a\} \cup FLOW(a)\} \cup IN \quad (3.5)$$

3.6 Loop Statement

Loop statement is a more complicated case than **if** statement. Consider the following example:

```
x = 1; a = 1;
for(i = 0; i < n; i++)
  x = x * a++;
```

Again, there is no direct flow from n to x , but after the loop execution, x is equal to $n!$. Another problem is the repeating execution of the loop body:

```
while (...)
{ a = b;
  b = c;
  ... }
```

One-pass analysis finds the flows $b \rightsquigarrow a$, $c \rightsquigarrow b$. If the loop body is executed more than once, the flow $c \rightsquigarrow a$ also exists. In the previous analysis of a statements block, we only considered *sequential flows*. Loop statements have the property of *cyclic flow*, i.e succeeding statements also affect preceding ones. Thus, for loops a two pass analysis is needed. Formally,

$$S \rightarrow \text{while}(E) S_1$$

$$IN(S_1) = \bigcup_{a \in E} \{\{a\} \cup FLOW(a)\} \cup IN, \text{analyze_twice } S_1 \quad (3.6)$$

where **analyze_twice** is a command to the Analyzer to scan the statement twice. A similar analysis is done for a **for** statement.

z	t
t	$\{z, _ \$1\}$

Table 1: Result of one pass analysis of loop

z	$\{t, _ \$1\}$
t	$\{z, _ \$1\}$

Table 2: Result FLOW table

3.7 Example

Let us consider the following example of a partial method's code, and analyze the flows occurring within it.

```
Copy(int x) { int z, t;
  z = 0; t = 1;
  while (t == 1)
  { z = z + 1;
    if (z == x)
      t = 0; } }
```

At the beginning the flows to local variables z and t are equal to \emptyset . The result FLOW of the first pass of the analysis of the **while** statement is shown in the Table 1. The flow IN of the loop body is variable t , then analysis of the first assignment statements finds $\text{FLOW}(z)=\{t\}$. The analysis of the **if** statement finds $\text{FLOW}(t)=\{z, _ \$1\}$.

The complete FLOW table of the method is shown in Table 2. The second pass analysis of the **while** statement, finds the flow $_ \$1 \rightsquigarrow z$. It is essential because during execution of the loop the value of the parameter x is copied into the local variable z .

3.8 Method Calls

In this section we discuss the analysis of methods calls. Since every method is analyzed independently, separate tables are created for each method. One table, called the CALLS table is used to store the flow into the called method parameters. The other table, called the RETURN table, contains the flow returned from a method call.

The CALLS table contains an entry for each method called from within the current method. Generally when a method A calls another method B , information may flow in both directions: from A to B via the Input or Input/Output parameters (note, we forbid the use of global variables...), and from B to A via the Output or Input/Output parameters or via the Return value. In this paper we restrict the discussion to *Input* parameters and *Return* value only, the case of Output and Input/Output parameters is discussed in [Gendler97].

Basically, an entry in the CALLS table is $(method, ParInFlow_1 \dots ParInFlow_n)$, where *method* is the name of the called method and *Pars* are sets of objects and virtual symbols that contain the information flow into the *method*'s parameters. If the called method returns a value via the Return statement, then this value is denoted also as a virtual symbol. Virtual symbols are used to denote information flow sets which are not known at Method compilation time and are instantiated only at *Transaction compilation time*. There are two kinds of virtual symbols: $_ \$i$ denotes information flow into parameter number i and $_ @j$ stands for

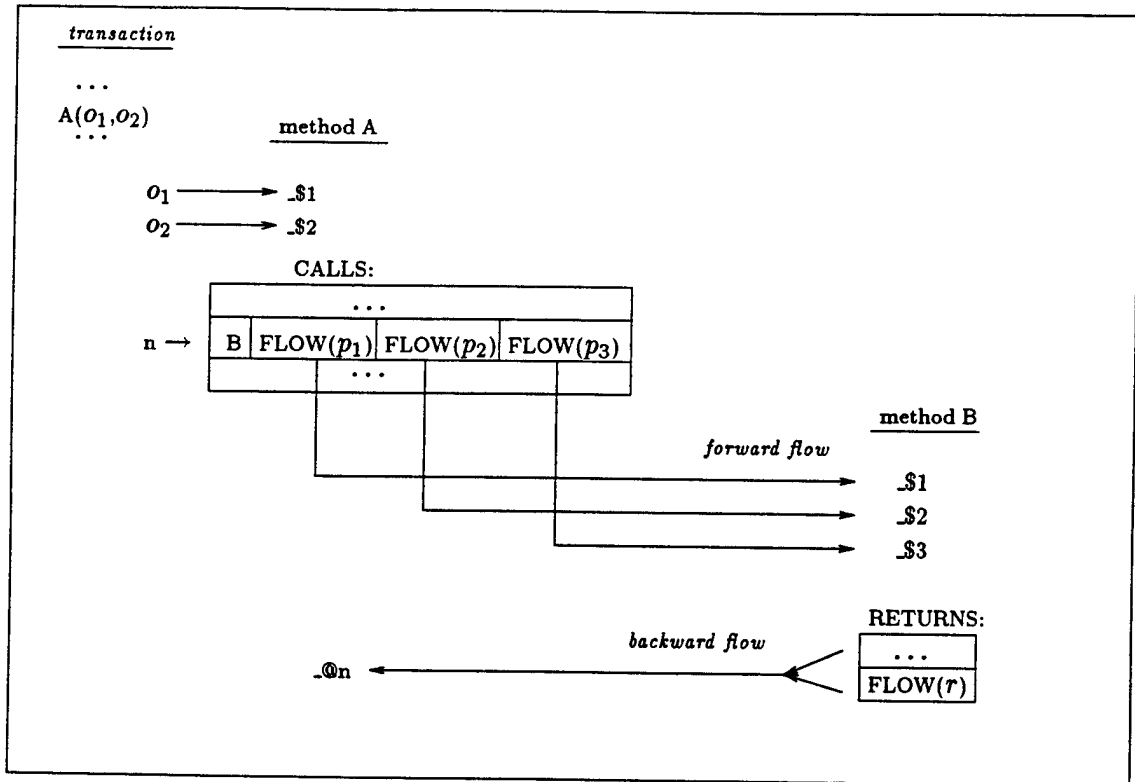


Figure 2: Analysis of transaction

information flow from the called method (via return value) and corresponds to entry number j in the CALLS table.

Assume now a transaction that invokes the method A. As the transaction is being compiled, the actual parameters passed to the methods are known. Let assume that real objects o_1 and o_2 , stand for the parameters of A and therefore, the virtual symbols $_{\$i}$ are *binded* to them. When B is processed, flows to the actual parameters substitute the virtual symbols $_{\$i}$ of B. The union of flows to all return values of B substitute the virtual symbol $_{@n}$ of A. Figure 2 demonstrates this process (see a full discussion in Section 5).

We now specify precisely the information that is entered into the tables. The first case is when the method is called within an arithmetic or another expression (i.e a function).

$$b = E \rightarrow \text{method}(E_1, E_2 \dots E_n)$$

The new entry in the CALLS table is:

$$(\text{method}, \text{GIFlow}(\bigcup_{a \in E_1} \{\{a\} \cup \text{FLOW}(a)\}) \dots \text{GIFlow}(\bigcup_{a \in E_n} \{\{a\} \cup \text{FLOW}(a)\} \cup \text{IN})) \quad (3.7)$$

where $\text{GIFlow}(a)$ is the set $\text{FLOW}(a)$ without local variables. To store the return value, e.g. within the flow of the expression E (which flows into b), the virtual symbol $_{@j}$ is used. A similar case is a call of the method which returns nothing.

3.8.1 Return value

The return value provides the means for backward information flow from the called method, to the method or the transaction which has called it.

$$S \rightarrow \text{return}(E)$$

An entry in the the RETURN table is inserted for each Return statement and it contains the following information flow

$$\text{GIFlow}(\bigcup_{a \in E} \{\{a\} \cup \text{FLOW}(a)\} \cup \text{IN}) \quad (3.8)$$

Note that since there may be multiple Return statements, several entries will be created, however, since we do not know which Return will be taken and we are interested in potential flow, these entries will be merged into a single entry at the Transaction analysis time.

4 Example

Consider the university database shown in Figure 1. Let us suppose that the salary of the university president is the most secret piece of information in the database. It is also reasonable to assume that the president of the university has the greatest salary between all university employees. We will investigate an example of illegal information flow caused by the called methods, in consequence of which the president's salary is compromised.

The methods code is shown below. In this code we have not used any special query language for read/write queries. Inside a method we just use the statement *read_object(o.A)* where *o* is a database object and *A* is an attribute. It is important to note that at Method analysis time, since in a statement such as *read_object(o.A)* the specific instance of *o* is not known, we use instead the root Class *O* of that object. In actuality, when the transaction analysis is performed, this root class is replaced with the restricted set of sub-classes and objects allowed access to the transaction's initiator (i.e the *AT.yes* subtree). In the tables, though we denote it as the root class, or as a virtual symbol.

The example below shows two methods, one calling the other. For each method we show the important tables generated by the method analysis phase.

```

Max_Payed_Employee(Employee_Hierarchy emp_tree)
{ int Max = 0;
  int ESSN, ESalary;
  Employee emp;
  for  $\forall emp \in emp\_tree$ 
  { ESalary = read_object(emp.Salary);
    ESSN = read_object(emp.SSN);
    if (ESalary > Max)
      Max = Store_Results(ESSN, ESalary)
  }
  return Max; }

```

```

int Store_Results(int res1, int res2)
{ Dummy dummy;
  write(dummy.val1, res1);
  write(dummy.val2, res2);
  return res2; }

```

First, as a result of the analysis of the method *Max_Payed_Employee*, the table *Max_Payed_Employee.FLOW* is generated as shown in Table 3. The first row shows the fact that local object *emp* contains the flow received via the parameter. The next two rows

<i>emp</i>	_\$1
<i>ESalary</i>	{_\$1.Salary, <i>emp</i> .Salary }
<i>ESSN</i>	{ _\$1.SSN, <i>emp</i> .SSN }
<i>Max</i>	{ <i>ESalary</i> , _\$1.Salary, <i>emp</i> .Salary, _@1 }

Table 3: Max.Payed_Employee.FLOW

Store_Results	{ _\$1.SSN, _\$1.Salary } , { _\$1.Salary }
---------------	---

Table 4: Max.Payed_Employee.CALLS

represent the flows caused by the read queries: The fourth row shows the information flow to local variable *Max* via the statement

$$Max = \text{Store_Results}(ESSN, ESalary) \quad (4.1)$$

This is the first case of a *method-call* discussed above. It is composed from the flow from the if statement and the flow returned by the called Method. The virtual symbol _@1 contains the flow returned from the Store_Results method. The flow into *Max* contains the IN flow and the flow resulted from the expression if (*ESalary* > *Max*), so all the information from the conditional expression flows to *Max*. It is *ESalary* and its FLOW which is {_\$1.Salary,*emp*.Salary }. That explains the three components of the entry for *Max* in the table.

Now, the calling of method Store_Results results with a new entry in table CALLS as shown in Table 4. The entry contains the name of the called method and its forward information flow - i.e. the information flowing into the parameters - *ESSN* and *ESalary* plus information flow to the statement (flow IN), recall that the statement is part of an if structure, so as seen above, { _\$1.Salary } is added to the flow of each of the parameters.

Now let us see the analysis of the method Store_Results. The FLOW table is empty and is not interesting. The WRITES table contains two entries as shown in Table 5. The RETURN table contains just the entry (_\$2). In the next section we shall see how the information contained in the above tables is combined for the detection of non-safe information flow.

5 Analysis of transactions

The analysis of a transaction is the final stage of the analysis described in this paper. Transactions may contain queries to the database - i.e. Read/Write queries, and calls to various methods. We assume that all components of a transaction are executed sequentially - i.e. no control flow statements are allowed. (this is not a real limitation since analysis similar to the one described for methods can be done, or alternatively they can be inserted within another

<i>Dummy.val1</i>	{_\$1}
<i>Dummy.val2</i>	{_\$2}

Table 5: Store_Results.WRITES

method.) In terms of Access Control, we assume the authorization model for methods in which a method corresponds to an Access Type, i.e internal actions of the method do not require additional authorization (see [GalOz93]).

The transaction analysis uses the auxiliary results of the methods analysis - i.e the tables WRITES, CALLS and RETURN discussed above. We add a new table called TR_FLOW to store all information flows caused by the transaction at any point of time. The table TR_FLOW will be used for computing the actual flows to the parameters of the methods. Note, that information flow has the property of transitivity: if flows $a \rightsquigarrow b$ and $b \rightsquigarrow c$ are safe, then the flow $a \rightsquigarrow c$ is safe too. This means that to verify safety of information flow $o_1 \rightsquigarrow o_2$ caused by writing within a method, there is no need to record all flows $o_i \rightsquigarrow o_1$ caused by previous write queries within the transaction and which were analyzed to be safe. In checking for safeness we use the ideas discussed in Section 2.3. That is, for Read Queries we first obtain by the authorization algorithm the subtree AT_yes and use it as the actual flow. The actual information flow is computed by *binding* the flow from the transaction with the virtual symbols recorded within the methods analysis tables. The analysis algorithm TranFlowControl is as follows:

```

TranFlowControl(transaction, initiator)
for  $\forall$  method meth invoked by the transaction
  switch (meth)
    case read query  $a = read(O.Attr_i)$ 
      Generate AT_yes using Initiator privileges
      TR_FLOW(a) = TR_FLOW(a)  $\cup$  AT_yes(query)
      break;
    case write query  $write(O.Attr_j, a)$ 
      Generate AT_yes using Initiator privileges
      if not Safe( TR_FLOW(a), AT_yes(O.Attr_j), initiator)
        return FALSE
      break;
    case method  $a = meth(p_1 \dots p_n)$ 
      if initiator  $\notin$  RACL(meth)
        break; /* no need to check flow */
      if not MethFlowControl(meth, TR_FLOW(p1)...TR_FLOW(pn))
        return FALSE
      TR_FLOW(a) = TR_FLOW(a)  $\cup$  ( $\cup$  meth.RETURN)
      break;
  return TRUE

```

The MethFlowControl algorithm works as follows: first, it substitutes the virtual symbols with real values, then it verifies all write queries within the method and checks for safe information flow. To verify the consistency of information flow in *all* invoked methods the algorithm works in a recursive manner. The decision about safety of a particular information flow is made during the process of the WRITES table binding:

```

MethFlowControl(m, fl1...fln)
  bind($1, fl1)
  ...
  bind($n, fln)
  for  $\forall$  entry i in CALLS table (meth, Par1...Parn)
    if not MethFlowControl(meth, Par1...Parn)
      return FALSE

```

```

    bind(_@i,  $\cup$  meth.RETURN) /*all entries taken */
    for  $\forall$  entry in WRITES table (Obj, WrFlow)
    if not Safe(WrFlow, Obj)
        return FALSE
    return TRUE

```

Note that the *bind* of Return value is possible, since it is known when we return from the recursion.

The **Safe** algorithms checks whether a given information flow is safe using the same algorithm as described in Section 2.3 (and in details in [GenGud97]). Remember, *CUAT(obj)* stands of the intersection of all objects which can be read by users who can also read object *obj*.

```

Safe(flow, obj)
    if Contain(flow, CUAT(obj))
        return TRUE;
    else
        return FALSE;

```

The proof of correctness for the above algorithms is quite obvious and is given in [Gendler97].

5.1 Example - continued

Let us illustrate the execution of the algorithm by the example of the method *Max_Payed_Employee* discussed in the previous section. Assume the following transaction which invokes the method *Max_Payed_Employee*.

```

begin transaction
1.create Private
2.emp_tree = read(Employee.{Salary,SSN})
3.a =Max_Payed_Employee(emp_tree)
4.write(Private,a)
end transaction

```

The object *Private* is the private object created by the initiator of the transaction. The AT_yes result of the read query is returned to *emp_tree*. This tree is a sub-tree of the Employee root class, and includes the objects permitted to the initiator of the transaction. (Note, different sub-trees may be authorized for different users). The method *Max_Payed_Employee* receives the tree as a parameter and returns the maximal payed employee within it. At the first glance the transaction seems legal. Even if the method will return the salary of the president (assuming the initiator has access to it), the initiator will write it to his private object which nobody has access to, and no non-safe information flow will occur. However, to see the whole picture we must analyze the information flows caused by the method *Max_Payed_Employee*. Applying the algorithm **TransFlowControl** we get the following results (refer to the numbered items in the transaction):

1. No flow recorded yet.
2. $TR_FLOW(emp_tree) = AT_yes(Employee.\{Salary, SSN\}, initiator)$
3. The flow from the method is computed by calling:
MethFlowControl(*Max_Payed_Employee*, $TR_FLOW(emp_tree)$).
and binding the parameters in the table *Max_Payed_Employee.FLOW*:
 $bind(_ \$1, TR_FLOW(emp_tree))$
Next, the following recursive call is made:

MethFlowControl(Store_Results, {TR_FLOW(*emp_tree*).SSN,
TR_FLOW(*emp_tree*).Salary}, {TR_FLOW(*emp_tree*).Salary})

The substitution of real data inside the virtual symbols is performed:

bind(_S1, {TR_FLOW(*emp_tree*).SSN, TR_FLOW(*emp_tree*).Salary})
bind(_S2, {TR_FLOW(*emp_tree*).Salary })

Now we have to analyze the WRITES table of this method. The decision about the safety of the information flow is made by the two calls to the algorithm **Safe**:

Safe({TR_FLOW(*emp_tree*).SSN, TR_FLOW(*emp_tree*).Salary}, Dummy.val1)
Safe({TR_FLOW(*emp_tree*).Salary}, Dummy.val2)

Clearly, the information flows:

{TR_FLOW(*emp_tree*).SSN, TR_FLOW(*emp_tree*).Salary} \leadsto Dummy.val1 and
{ TR_FLOW(*emp_tree*).Salary } \leadsto Dummy.val2 are detected.

4. The flow into the object *private* is computed, but since this is a private object, its CUAT is at least *AT_yes*, therefore this write is obviously safe.

The safeness of this transaction is therefore dependent on the safeness of step 3. Let us imagine that Dummy is a public object open to all users (i.e. we can assume that CUAT(Dummy) may contain very few objects, maybe the salary of the students employees only). So the safety of the information flow depends on the privileges of the transaction initiator *u*. If the transaction initiator is a user allowed to read students' salaries only, i.e.

$$\text{TR_FLOW}_u(\text{emp_tree}) = \text{Student}.\{\text{SSN}, \text{Salary}\}$$

then there is no non-safe information flow. But, if the transaction initiator is a user allowed to read the president's, i.e.

$$\text{TR_FLOW}_u(\text{emp_tree}) = \{\text{Manager}, \text{President}\}.\{\text{SSN}, \text{Salary}\}$$

then the method *Max_Payed_Employee* plays the role of a **Trojan Horse** [Samar97]. Clearly, this illegal flow is discovered, since TR_FLOW is not contained in CUAT(Dummy).

5.2 Authorization Rules Maintenance

As was explained in Section 2.3 and used in the **Safe** algorithm, the main data structure with which the flow computed at compile-time is checked, is the structure: *CUAT* - the common user access tree. This structure can be uniquely determined for a given set of authorization rules. However, when authorization rules change, this structure has to be recomputed.

There are basically two approaches. One is to recompute it every time a new transaction is compiled. This carries heavy computational overhead but saves storage. The other option is to compute it once and stores it for each object (class), and maintain it when authorization rules are added or deleted. Since the events of changes in authorization rules are much less frequent, this is much better computationally. Algorithms to maintain the CUAT structure were also presented in [GenGud97]. A similar approach is advocated by [Bertino(96)].

6 Conclusions

In this paper we discussed the problem of protecting against unsafe information flow in object-oriented databases. Previous models for such protection provide a run-time mechanism which carries a considerable run-time overhead. This paper presents a compile-time

solution to the problem. It relies on the compile-time analysis of methods which uses well known program analysis techniques. It then uses the Methods analysis phase in analyzing the transactions and deciding at transaction compile-time whether an unsafe information flow exists or not. In the future we like to consider the case of distributed transactions (or autonomous objects) when no single-centralized control is available for the analysis phase.

References

- [Aho86] A.V.Aho,R.Sethi,J.D.Ullman *Compilers, principles, techniques and tools*, Addison-Wesley, 1986.
- [Bertino(96)] Bertino, E., Bettini, C., Ferrari, E., Samarati, P., "A Temporal Access Control Mechanism for Database systems," *IEEE Trans. on Knowledge and Data Engineering*, Vol 8, No. 1, pp. 67-80.
- [Cast95] Castano, S., M. Fugini, G. Martella, P. Samarati, *Database Security*, Addison-Wesley, 1995.
- [Denn86] D.E.Denning *Cryptography and Data Security*, Addison-Wesley, 1983.
- [GalOz93] N.Gal-Oz, E.Gudes and E.B.Fernandez "A Model of Methods Access Authorization in Object-Oriented Databases.",*Proc. of the 19th VLDB Conference*, Dublin,Ireland,1993.
- [GenGud97] M.Gendler-Fishman,E.Gudes *A Compile-time Model for safe Information Flow in Object-Oriented Databases*,Information Security in Research and Business, Edited by L. Yngstrom and J. Carlsen, Chapman & Hall, 1997, pp. 41-55.
- [Gendler97] M.Gendler-Fishman *A Compile-time Model for safe Information Flow in Object-Oriented Databases*,thesis for MSc, Ben-Gurion University.
- [Griff76] Griffith, P., Wade B., "An Authorization Mechanism for a Relational Database System," *ACM Trans. on Database Systems*, Vol 1, No. 3, September, 1976.
- [Fern94] E.B.Fernandez, E.Gudes, H.Song "A Model for Evaluation and Administration of Security in Object-Oriented Databases.",*IEEE Trans. on Knowledge and Data Engineering*, Vol.6. No.2., April 1994, pp. 275-292.
- [Kemp94] Kemper A., G. Moerkotte, *Object-oriented Database Management*, Prentice-Hall, 1994.
- [Kim90] Kim W., *Introduction to Object-Oriented Databases*, The MIT Press, 1990.
- [Larr90] Larrondo-Petrie M., Gudes E., Song, H., Fernandez E B., "Security Policicies in object-oriented databases," *Database Security IV: Status and Prospectus*, D. L. Spooner C. E. Landwehr (Ed.), Elsevier Science Publishers, 1990, pp. 257-268
- [Muchnick81] S.S.Muchnick,N.D.Jones *Program Flow Analysis: theory and applications*, Prentice-Hall,1981.
- [Samar97] Samarati P., E.Bertino, A.Ciampichetti and S.Jajodia "Information Flow Control in Object-Oriented Systems," to appear in *IEEE Trans. on Knowledge and Data Engineering*, 1996.
- [Stone76] Stonebraker, M., Wong, E., Kreps, P., Held, G., "The Design and Implementation of Ingres", *ACM Trans. on Database Systems*, Vol 1, No. 3, September, 1976.

Capability-Based Primitives for Access Control in Object-Oriented Systems

J. Hale J. Threet S. Shenoi

Department of Mathematical and Computer Sciences
The University of Tulsa
Tulsa, OK, 74104-3189

Abstract

Access control is the cornerstone of information security and integrity, but the semantic diversity of object models makes it difficult to provide a common foundation for access control in object-oriented systems. This paper presents a primitive capability-based access control architecture that can model a variety of authorization policies for object-oriented systems.

The architecture described is integrated at the meta-object level of the Meta-Object Operating System Environment framework, providing a common foundation for access control in heterogeneous object models.

1 Introduction

Access control is of critical importance to the security and integrity of multiuser distributed systems, from distributed databases, local networks and intranets to the Internet itself. While object-oriented technology has become a touchstone for developing modern distributed systems, the full potential of access control mechanisms for objects has yet to be realized. Most of the advances and successful applications of object access control have been confined to the area of database security [5, 6, 13, 19, 20, 26].

Several factors have limited the incorporation of access control mechanisms in mainstream distributed object systems technology. Object models are heterogeneous in nature with tremendous semantic diversity. Authorization policies, which are greatly influenced by the specific object models they are designed to protect, cannot be applied to other object models. Moreover, most access control mechanisms are brittle, incapable of supporting multiple policies.

The Meta-Object Operating System Environment (MOOSE) being developed at the University of Tulsa provides a framework for the development of verifiably secure heterogeneous distributed objects with a combination of formal methods and object technology [16]. This paper describes a capability-based access control architecture for distributed objects. This architecture

extends an earlier incarnation of the MOOSE framework with a "meta-level integration" of primitive security mechanisms that can be used to implement a variety of authorization models.

Object-oriented systems share a common theoretical underpinning – the notion of a meta-object. MOOSE uses the meta-object concept to model diverse object-oriented systems and to provide a foundation for secure interoperability. The Meta-Object Model (MOM) in MOOSE expresses diverse object-oriented features using a few core mechanisms. By integrating security mechanisms into MOM, it is possible to capture a variety of authorization models for object-oriented systems. In particular, Access Control Lists (ACLs) and access monitors are integrated to permit the modeling of various forms of Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role-Based Access Control (RBAC).

Capabilities provide a useful mechanism for reconciling the inherent differences between identity-based DAC, label-based MAC and RBAC. Capabilities are unforgeable tokens that give the possessor certain rights to an object [11, 21, 22]. Traditionally, they have been used to keep track of access rights at runtime. The flexibility of capabilities suggests their use as a meta-model for access control to concurrent objects operating in distributed environments.

This paper proposes a capability-based access control architecture for meta-objects as a common foundation for security in heterogeneous distributed object systems. The paper begins with an overview of access control in object-oriented systems which also motivates the work. The Meta-Object Model (MOM) and the access control architecture are described in detail along with their roles in capturing heterogeneous access control models. The paper concludes with a comparison of related work and future research directions.

2 Access Control of Objects

Object-oriented systems are composed of classes, instances, attributes (instance variables) and methods. These components support encapsulation, modularity and re-use through message-passing, inheritance and aggregation.

The goal of access control is to protect resources (objects) from unauthorized access by users (subjects). An *authorization state* is a function $State : (Subject \times Object \times Privilege) \Rightarrow Boolean$, where the subject's privilege is the access type. Often, the authorization state is represented as a list of authorization tuples, e.g., $\langle Subject, Object, access_type \rangle$. Each tuple declares that *Subject* has the *access.type* privilege for *Object*. An access control model defines domains for subjects, privileges and objects. Also, it defines rules for implicit authorization and specifies a set of commands to take systems from one authorization state to another.

Although access control is fundamental to information security and integrity, authorization models in object-oriented database management systems (OODBMSs) [23, 24, 27] lack a common perspective compared with those for relational database management systems (RDBMSs). In fact, many OODBMSs do not provide any type of access control. The semantic diversity of object models is partly to blame. For example, some object models support multiple inheritance, while others do not. The presence of competing authorization models also introduces problems. Each model addresses protection granularity, access types and implicit authorization flow for objects in its own special way. The dilemma is illustrated using a simple electronic commerce example.

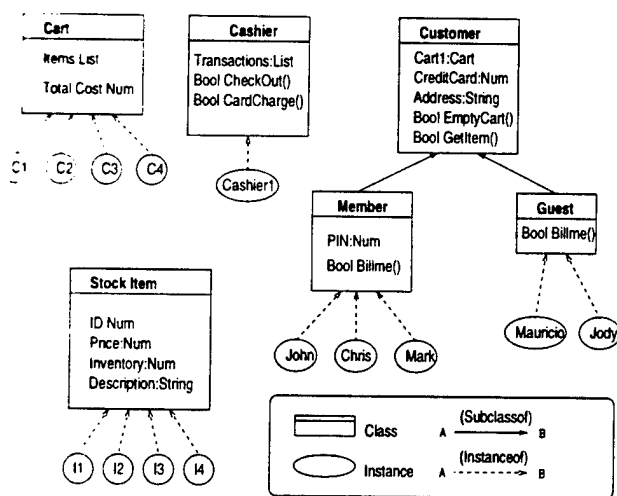


Figure 1. Object-oriented electronic commerce model.

Consider an object-oriented model of an electronic commerce system shown in Figure 1. The protection granularity specifies the finest units to be protected. In relational databases, these range from tables to tuple elements. Classes, objects, attributes and methods are all viable atomic units for protection in an object-oriented system.

Objects could be chosen as the finest unit of protection in this example. While the resulting "all or nothing" access to objects is efficient (only one authorization set is needed per object), the approach is rather inflexible. In a real-world scenario, a customer must be able to execute `Cashier::CheckOut()` to pay for items. Using objects as atomic protection units implies that Customer objects would have access to Cashier objects. But this would allow Customer objects to access a potentially sensitive list of Cashier transactions as well. The inflexibility of this approach forces most object models to respect individual methods and attributes as atomic units of protection, e.g., permitting a Customer access to `Cashier::CheckOut()`, but denying her access to `Cashier::Transactions`.

Access types are another major issue in object-oriented systems. Database authorization models make use of *read* and *write* permissions. The privilege of modifying rights can introduce access types (e.g., *grant* and *revoke*), although in many systems this is a power implicitly held by owners of objects. Providing *grant* and *revoke* types has some disadvantages. For example, the ability to *grant* can be an access type (*grant-grant*) in its own right. Implementing this feature usually requires a self-referential access type.

Suppose that cashiers need to read customers' addresses, but should not be able to modify them. Authorizing $\langle \text{Cashier1, Jody}::\text{Address}, \text{read} \rangle$ only gives `Cashier1` read access to `Jody::Address`. If attributes can be read or written directly (without using a local accessor method), access types *read* and *write* are desirable. However, if the object model relies on local accessor methods to preserve encapsulation, then separate methods should exist for reading and writing attributes and effective read/write control can be manifested via the *execute* privilege on those accessor methods. The *execute* type for method-based access control was proposed by Fernandez *et al.* [15].

Implicit authorization is a convenient way of propagating permissions and protections in a large environment. The idea is that permissions/protections can be given to an entire class of subjects/objects using one authorization rule. The application of this no-

tion to object-oriented systems is most natural because the *class* concept is fundamental to most object models. An instance can inherit its authorization status from its class in the same way that it inherits methods and attributes. The authorization rule $\langle \text{Customer}, \text{StockItem}::\text{Price}, \text{read} \rangle$ gives all customers access to the prices of all items in stock.

Generating *exceptions* to authorization rules is possible through the use of *negative authorizations* and *strong/weak authorizations*. A negative authorization explicitly denies a subject access to an object. For example, $\langle \text{Chris}, \text{Cashier}::\text{CheckOut}(), \neg \text{execute} \rangle$ explicitly prohibits Chris from checking out. Obviously, the mixture of positive and negative authorizations can lead to conflicts in an authorization model. Therefore, authorization in such schemes is usually derived by labelling rules as either *weak* or *strong*. Strong rules cannot be overridden by weak ones. While the presence of such rules makes it difficult to derive an authorization rule for a particular event, they allow for highly expressive authorization models.

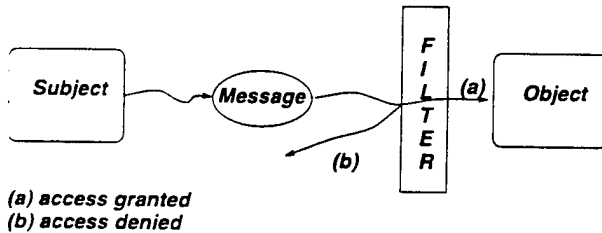


Figure 2. Message filter.

Messages are the principal medium of communication in object-oriented systems. Jajodia *et al.* [19] were the first to propose that messages be used as the focus of access control mechanisms in object systems. Towards this end they introduced a *message filter* for deciding whether or not to accept a message on behalf of an object based on its source, content and destination (Figure 2). A message filter can be made to reside within each object, providing ubiquitous access control in an object-oriented system. This decentralized authorization technique is superior to and more natural than centralized access control schemes for distributed object systems.

The approach to access control of objects described in this paper relies on a decomposition of object systems into their most primitive components. Since message-passing is central to any meta-object model, the message filter presents itself as an integral piece of an architecture for unifying access control in heterogeneous distributed object systems. The following section explores access control for meta-object models

and presents a flexible authorization architecture that is easily integrated with existing object models.

3 Meta-Object Access Control

This section describes a primitive operational foundation for access control in object-oriented systems. It is designed to be integrated at the meta-object level to permit a unified treatment of meta-classes, classes and objects [28]. The main requirement of the meta-level access control model is that it be flexible enough to support multiple access control policies in distributed computing environments.

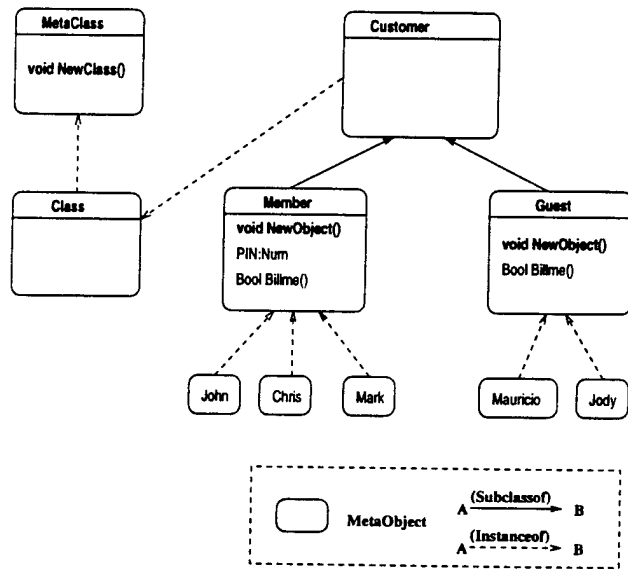


Figure 3. Meta-object electronic commerce model.

Meta-object models provide a common theoretical underpinning for object-oriented systems. They are capable of presenting a unified view of object-oriented features such as classes, subclasses and inheritance with more primitive notions of meta-objects and delegation. Meta-object models can integrate method invocation, message-passing and aggregation, the basic features of all object systems.

Figure 3 illustrates a decomposition of the object-oriented electronic commerce model into a meta-object representation. Note that all classes and objects have been replaced by meta-objects. Meta-objects that can spawn other meta-objects (using a method called *NewObject()*) model classes. Furthermore, meta-objects capable of spawning "class" meta-objects are meta-classes. *MetaClass* and *Class* are both meta-classes.

This model uses capabilities [11, 21, 22] to provide method-based access control for meta-objects. A capability is an unforgeable token that a subject uses

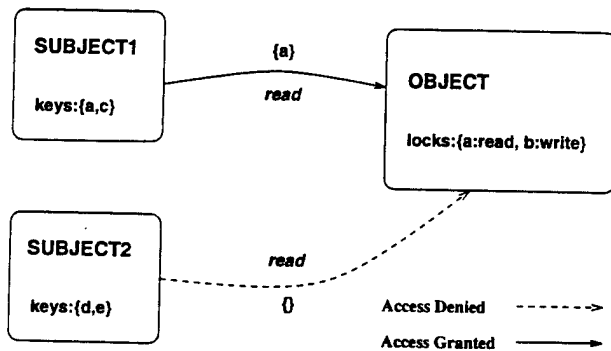


Figure 4. Access control with capabilities.

as a ticket for object access. A ticket, which is also associated with an access type, can be held by a subject or for a subject by a trusted third party. The ticket is inspected by an object or by the trusted third party before access is granted. Alternatively, capabilities can be viewed as *locks* and *keys*; this view implies that objects must hold matching tokens. Each subject has a number of keys that give access to objects with matching locks. This analogy is illustrated in Figure 4. SUBJECT1 has key *a* that matches a *read* lock held by OBJECT and, therefore, is given *read* access to OBJECT. The intersection of SUBJECT2's keys and OBJECT's locks is empty. Therefore, SUBJECT2 has no access to OBJECT.

Capabilities typically control run-time privilege distribution between processes and subprocedures. They can be used to implement various authorization models, including identity and group-based Discretionary Access Control (DAC), Role-Based Access Control (RBAC) and Mandatory Access Control (MAC). This flexibility makes capabilities ideal for meta-level access control and conducive to supporting multipolicy functionality.

In general, an authorization model must resolve (*s, o, a*) as true or false for every subject (*s*), object (*o*) and access type (*a*) given the authorization state. An authorization state is defined by $State : Object \rightarrow Privilege \rightarrow Token \rightarrow Bool$ where *Token* serves as a representative for the subject.

Recursive definitions are used to specify allowable access types in the primitive access control model. This is necessary to implement a general model of grant and revoke privileges. The recursive access types are defined in Figure 5. The type definition permits access types such as G. G. LOCK (grant grant lock) and G. R. G. KEY (grant revoke grant key). These types can be thought of as pertaining to token lists. Note that every type other than KEY behaves as a lock,

e.g., G. R. KEY is a type of lock that guards the R. KEY list. Any subject with KEY token matching a token associated with G. R. KEY in an object can add (grant) tokens to the R. KEY list.

ACCESS TYPES

```

priv ::= ALL
      | priv2

priv2 ::= KEY
      | LOCK
      | G . priv2
      | R . priv2

```

ACCESS CONTROL COMMANDS

```

comm ::= ADD -> priv -> token -> object -> object
      | REMOVE -> priv -> token -> object -> object
      | comm ; comm

```

ACCESS CONTROL PREDICATES

```

EVAL: comm -> state -> state -> bool

TRANS: state -> state -> bool

```

Figure 5. Access control definitions.

The ALL privilege in Figure 5 confers privileges of every type to a subject. A subject carrying a token with the ALL privilege on some object can add or remove any type of authorization. The only limitation is that the subject must hold the actual token as a key. Rule 1 in Figure 6 formalizes the semantics for the ALL access type.

The command set for a user enables alteration of the authorization state. This set comprises commands for adding and removing authorization tuples to and from objects. The command set is given in Figure 5. Subjects can only add or remove tokens that it holds as keys. Taking the list-based view, this means that even when a subject has a grant or revoke permission on some access control list, the only tokens it is able to add or remove are those that it holds as keys. The third command allows sequencing of ADD/REMOVE commands.

Two predicates, EVAL and TRANS, are defined on authorization states (Figure 5). EVAL returns true when a command will take one state to another. It is used to define the semantics of the command set. TRANS returns true if a transition between states is possible. The relationship between EVAL and TRANS is formalized by Rule 2 (Figure 6).

Rule 1 : $\forall p : priv, o : obj, t : token, s : state.$
 $s \circ ALL\ t \Rightarrow s \circ p\ t$

Rule 2 : $\forall s_1, s_2. \exists c : comm.$
 $EVAL\ c\ s_1\ s_2 \Rightarrow TRANS\ s_1\ s_2$

Rule 3 : $\forall p, s_1, s_2, o_1, o_2, t.$
 $s_1\ o_1\ KEY\ t \wedge s_1\ o_2\ G.p\ t \Rightarrow$
 $(s_2\ o_2\ p\ t \wedge (\forall o', p', t'. o' \neq o_2 \vee p' \neq p$
 $\vee t' \neq t \Rightarrow s_1\ o' p' t' = s_2\ o' p' t')) \Rightarrow$
 $EVAL\ (ADD\ p\ t\ o_2\ o_1)\ s_1\ s_2)$

Rule 4 : $\forall p, s_1, s_2, o_1, o_2, t.$
 $s_1\ o_1\ KEY\ t \wedge s_1\ o_2\ R.p\ t \Rightarrow$
 $\neg(s_2\ o_2\ p\ t \wedge (\forall o', p', t'. o' \neq o_2 \vee p' \neq p$
 $\vee t' \neq t \Rightarrow s_1\ o' p' t' = s_2\ o' p' t')) \Rightarrow$
 $EVAL\ (REMOVE\ p\ t\ o_2\ o_1)\ s_1\ s_2)$

Rule 5 : $EVAL\ (c_1)\ s_1\ s_2 \wedge (c_2)\ s_2\ s_3$
 $\Rightarrow EVAL\ (c_1; c_2)\ s_1\ s_3$

Figure 6. Access control rules.

Rule 3 supplies formal semantics to the **ADD** command. This rule states that a subject must have grant privilege over an access type in an object to add a token of that type to the object. The constraint that subjects can only add tokens held by them as keys is formalized. For example, authorization tuples $\langle o, G.R.LOCK, a \rangle$, $\langle s, KEY, a \rangle$ and $\langle s, KEY, b \rangle$ allow the command **ADD R.LOCK b o s**.

The semantics for the **REMOVE** command (Rule 4) specify when it is legal for a subject to remove authorization tuples. Using the previous example, an additional authorization tuple $\langle o, R.R.LOCK, b \rangle$ would let s remove **R.LOCK** permissions from o . Again, s must hold the affected token as a key.

Rule 5 introduces command sequences to the system. It formalizes the transitive nature of commands on authorization states.

4 Authorization in the Meta-Object Model

The Meta-Object Model (MOM) in MOOSE is a core model for the design and analysis of sophisticated distributed object systems. MOM is augmented with primitive mechanisms intended to support a spectrum

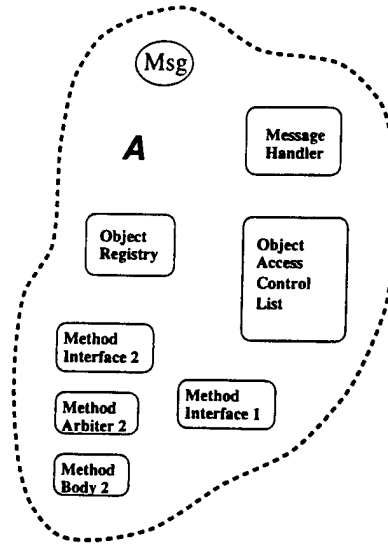


Figure 7. MOM object components.

of authorization models for object-oriented systems. MOM will permit developers to construct and analyze secure object-oriented programming languages and database management systems. This section describes MOM and the integration of capability-based access control primitives in MOM.

MOM is defined with the Robust Object Calculus (ROC), a process calculus tailored to modeling distributed objects [30]. The underlying principles of ROC, e.g., encapsulation and tuple-based communication, have facilitated the formal design of MOM. The design of MOM is influenced by ACTORS [1] and Concurrent Aggregates [7]. MOM supports core object functionality, including persistence, method invocation, asynchronous message-passing, delegation and aggregation. Virtually any object model can be constructed from this core functionality.

Access control policies are modeled in MOM systems with object access control lists (OACLs) and message filters. These capability-based mechanisms implement flexible and ubiquitous access control for objects and methods. Objects with OACLs and message filters can provide authorization services to domains of subobjects when efficiency concerns outweigh security requirements.

4.1 MOM Objects

MOM objects are viewed as a collection of tightly encapsulated components (processes). Each MOM object has a set of identifiers that defines how it can be addressed. MOM components that share an identifier are said to be part of the same object. Identifiers in MOM are navigational tuples of names. Navigational

identifiers (nids) are defined using the following BNF rules.

```

nid ::= [olid#,nid#] | nil
olid ::= [direction,lid#]
direction ::= in | out

```

A MOM system entails a hierarchical structure of object domains, specifying objects that contain objects. Every MOM system contains exactly one *root* object, which is not contained by any other object. The root object is used in a bootstrapping process that initializes MOM systems. Each object is named by a local identifier (lid) unique to its domain. The domain of an object is its parent object.

The local identifier of an object (say Obj_1) is prepended to the global identifier (gid) of its parent (say *root*) to define a unique gid for the object, e.g., $[out, Obj_1\#] \# [out, root\#] \# nil\#] \#$. Objects consist of a number of cooperating MOM components: an *object registry*, a *message handler* and *methods*. The message handler and the object registry form the basis for an object's identity, controlling communication for a tightly bound set of components. Furthermore, objects can house a message filter and an *object access control list* (OACL), which contains authorizations for access to the object's components. The message filter resides in the message handler and authorizes each message using the OACL. MOM object components, including the OACL and message filter, are illustrated in Figure 7.

An object registry maintains a record for each active component within the object. In particular, an object registry keeps track of message handlers, method interfaces and each active method invocation. The lid, the component type and a third element containing miscellaneous information are stored within each registry record. As an object is being deleted it must refer to its registry so that it can gracefully delete each of its components. Furthermore, to create an object inside a parent, the registry is checked to see that the new lid is unique. Only then is the object created and registered with the parent object.

4.2 MOM Message-Passing

Messages embody asynchronous communication in MOM. Messages carry method invocation requests, acknowledgements and replies between objects. Once a request is sent, the sender can continue its computation without waiting for an acknowledgement or reply, i.e., all communication is asynchronous. MOM messages are categorized as method invocation requests (**request**), subsequent replies (**reply**) or acknowledgements (**ack**).

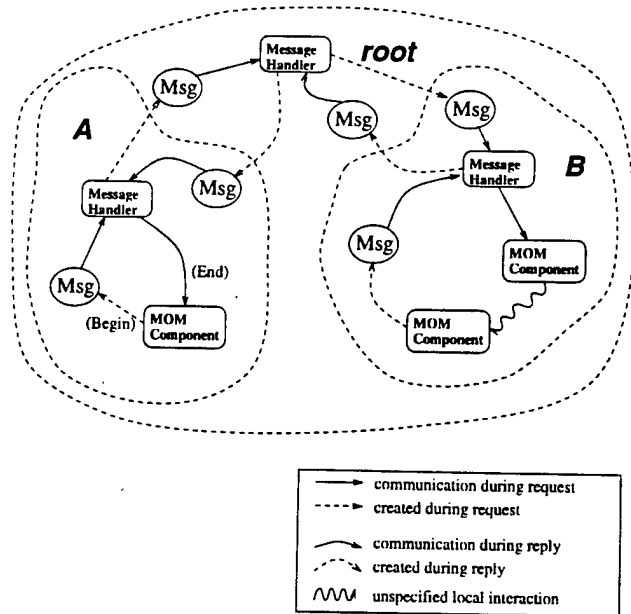


Figure 8. Messaging in MOM.

A message handler processes incoming MOM messages and marshalls object requests. It controls the distribution of requests and replies for a tightly bound set of components. Figure 8 illustrates the creation and acceptance of messages by message handlers for a request/reply sequence.

An incoming message can be received as a local method invocation request or delegated to another object. A message handler "delegates" a message by consuming it and re-creating it in an adjacent domain. For example, the root domain's message handler M_Hndlr_{root} consumes the reply message $Msg_{B \rightarrow root}^{rep}$ and creates a new message $Msg_{root \rightarrow A}^{rep}$ in A's domain.

4.3 MOM Methods

The method architecture in MOM is composed of three types of communicating processes *method interfaces*, *method arbiters* and *method bodies*. *Method interface* components accept invocation requests propagated by message handlers. These interfaces control access and synchronization for individual methods. A unique method interface exists for each method in an object.

Upon receiving an invocation request, a method interface will spawn a method arbiter and method body. The method body process performs the actual computation, while the method arbiter negotiates communications between the method body and the outside world. The MOM method subsystem is shown in Figure 9.

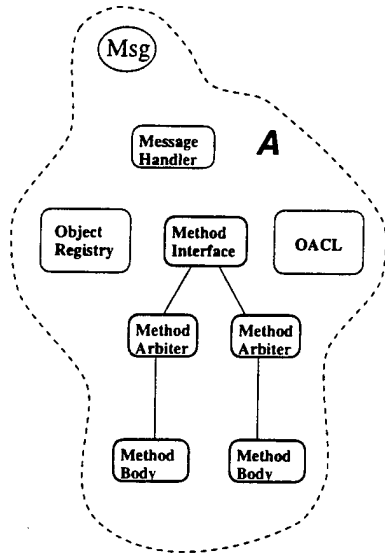


Figure 9. MOM method scheme.

The MOM toolkit has *mutable* and *immutable* methods. A mutable method invocation spawns a persistent process with state that can be accessed many times. Immutable methods, on the other hand, behave like traditional methods, terminating and returning a value upon completion. While mutable methods model instance variables, it is a MOM standard to create (immutable) accessor methods in the object for instance variables.

Requests to an immutable method spawn a new method arbiter and body to support concurrent method invocation. This may or may not be allowed by the method interface which controls synchronization. A mutable method interface does not permit co-existing invocations. Only one body and arbiter can be active at a time. Requests to an active mutable method are forwarded to the method arbiter. All requests for a mutable method are forwarded to its arbiter which then forwards it to the method body. The behavior of the method body can be affected by previous requests; this models state information.

Immutable methods behave like methods in conventional object-oriented programming languages. Each request spawns a new method body supporting concurrent method invocation. The completion of an invocation results in a reply from the method to the method arbiter. This reply can be propagated back to the initiating object (modeling a traditional method invocation) or it can go elsewhere as indicated by the request.

Method interfaces for immutable methods accept invocation requests and create new method arbiters

and bodies for each request. Method arbiters manage individual method bodies. For immutable methods this entails passing parameters to a method body and waiting for a reply. Method bodies are also defined with the restriction that they be properly encapsulated to avoid interference with other MOM components. A method body, once created must receive a request sent by its parent arbiter. A method body can in turn invoke other methods, including those in foreign objects.

An immutable method must formulate an appropriate reply to complete an invocation. A parameter passed to the method body determines the correct type of reply; this can be *noreply* if no reply is necessary, *nil* to receive a normal reply, or it could be the *mid* of another object.

Component creation and deletion is handled by special methods, *NewMethod*, *NewObject*, *DeleteMethod*, and *DeleteObject*, that are resident in each object. Objects invoke these methods the way they would any other method. For example, a foreign object might issue a request to another object to create a subobject. However, methods might refuse this request if a conflict exists in the object registry (e.g., a subobject of the same name exists) or even as a matter of policy (e.g., only ancestors can delete object components).

4.4 MOM Security

OACLs and message filters implement access control in MOM systems. Each meta-object can contain an OACL and a message filter. Each OACL contains a list of authorization tuples of the form *< component, accesstype, token >*. Message filters use these lists to authorize messages received by message handlers. A message contains source, destination and content information. The content information specifies the type of access, e.g., *G. LOCK*, as well as a set of tokens provided by the message originator to be used as keys in the authorization process.

Method-based access control is specified by authorization tuples that contain method names in the component field. Authorization commands, e.g., *G. KEY*, can also be issued in messages to destination objects. However, filtration can occur at the destination and at all objects along the message route. Therefore, delegating a message between objects must be authorized. This feature protects entire objects.

A message filter examines a message and compares it with its OACL to determine authorization. If the message source holds a key matching a lock held by the intended component recipient, the message is authorized. This simple scheme functions in the realm of object-oriented programming languages and is par-

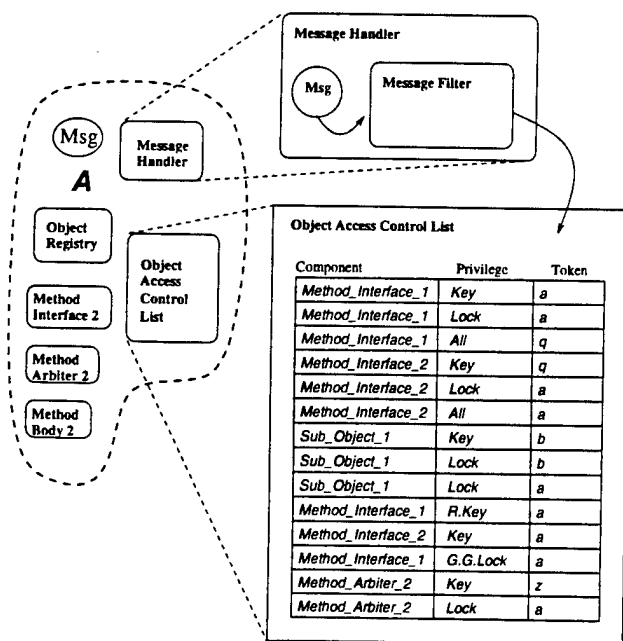


Figure 10. OACL and message filter components.

ticularly well-suited to object-oriented databases and distributed object systems. Figure 10 shows the components of a MOM object with the OACL and message filter.

MOM objects affect the access control scheme by introducing intervening filters. Assume that object *s* has legitimate access to object *o* by virtue of *s* having a key matching one of *o*'s locks. Object *s* could be denied access to *o* if an intervening object *i* exists on the message path between *s* and *o*. This can occur because messages must be authorized to pass through each domain between source and destination. If *s* is not allowed to send messages to *i*, the message will be returned at that point and access to *o* will be effectively denied. Intervening objects complicate the authorization architecture but facilitate specialization of authorizations.

The root object contains all other objects and plays an important role in the bootstrapping process by creating meta-objects and initializing the authorization state. Once meta-objects are created they in turn spawn objects and propagate authorizations when necessary. Classes are meta-objects with special methods that construct instance objects of the same type. Authorizations can be inherited by instances or subclasses via token propagation and runtime delegation. These techniques manifest implicit authorization flow in object-oriented systems. User-controlled objects manifest explicit authorization at runtime by invoking

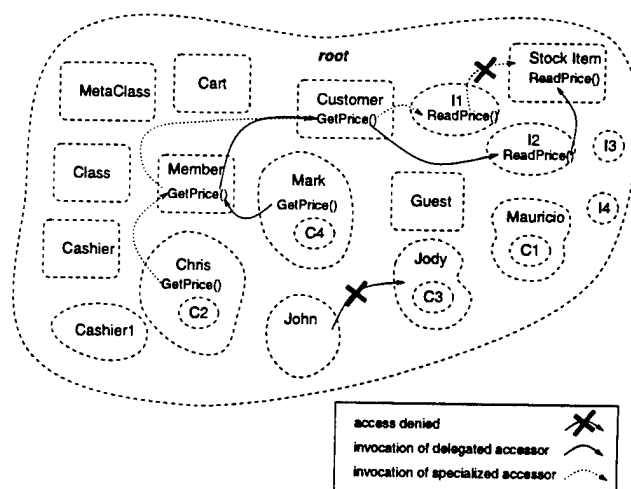


Figure 11. MOM electronic commerce.

ing methods containing authorization commands.

Consider the MOM representation of the electronic commerce model illustrated in Figure 11. It is complemented by a partial view of the virtual global OACL in Figure 12. The global OACL shows the authorization state after the bootstrapping process.

No customer should be able to access the information of another customer. This constraint is enforced by ensuring that no instance of *Customer* has a key that matches a lock in the OACL of the root object (which contains all customer instances). Figure 11 shows the result of John trying to access Jody. Note that access to Jody and its subobjects is denied because the filter in root intervenes. The filter checks the OACL in root and discovers that John does not have permission to send messages to (or through) Jody.

Another reasonable policy is that customers should be able to read, but not modify, stock item prices. Therefore, the authorization tuple `< ReadPrice(), LOCK, read_price >` should be contained in the OACL for the meta-object (class) *Stock Item*. Customer access to prices through *Customer::ReadPrice()* implies that the OACL for meta-object (class) *Customer* contains the authorization tuple `< ReadPrice(), KEY, read_price >`. The authorization commands to add these tuples are given by the root object in the bootstrapping process.

Delegations must be authorized in the same way that method invocations are authorized. Meta-objects are responsible for propagating delegation authorizations to instances. *Stock Item* must propagate `< ReadPrice(), LOCK, read_price >` to each of its instances and/or subclasses. This prop-

Object	Component	Privilege	Token
Stock Item	MsgHandler	LOCK	cashier
I2	MsgHandler	LOCK	cashier
I3	MsgHandler	LOCK	cashier
I4	MsgHandler	LOCK	cashier
Stock Item	ReadPrice()	LOCK	read_price
I2	ReadPrice()	LOCK	read_price
I3	ReadPrice()	LOCK	read_price
I4	ReadPrice()	LOCK	read_price
root	Customer	LOCK	cashier
root	Stock Item	LOCK	cashier
root	Stock Item	LOCK	customer
Customer	MsgHandler	KEY	customer
John	MsgHandler	KEY	customer
Customer	ReadPrice()	KEY	read_price
John	ReadPrice()	KEY	read_price
Customer	MsgHandler	LOCK	cashier
John	MsgHandler	LOCK	cashier

Figure 12. Global view of OACLs.

agation is performed in the constructor `StockItem::NewObject()`.

When an instance of `Customer` reads a price, it delegates up to `Customer::ReadPrice()` which contains the correct key for authorization to stock item prices. In particular, Figure 11 shows the method invocation chain when `Mark::GetPrice()` is invoked for I2. At the other end of this method invocation request, `Customer::ReadPrice()` invokes an accessor function local to I2, which is then delegated to an accessor method in `Stock Item`.

Authorizations for method-based access can be specialized in the same manner that methods are specialized. The authorization to delegate is held by the delegating object (the instance or subclass), facilitating authorization specialization. For example, the accessor method in I1 could choose not to delegate to its parent class, but instead define its own behavior and/or authorization set. This is illustrated in Figure 11 where Chris attempts to get the price of I1, but is denied because the authorization tuple $\langle \text{ReadPrice()}, \text{LOCK}, \text{read_price} \rangle$ for delegation to the parent class is missing (Figure 12).

Efficiency is a primary concern when message filters are used to implement access control. The proposed architecture permits objects to provide authorization services for entire domains. This obviates the use of message filters and OACLs for each and every object in MOM, resulting in a flexible and potentially lightweight access control system.

5 Related Work

The approach taken by this work is motivated by the object systems LOOPS [28] and ACTORS [1]. Both systems probed the foundations of the object paradigm and developed sophisticated meta-object models. The meta-level access control mechanisms embedded in the Meta-Object Model (MOM) described in this paper support multipolicy access control in heterogeneous object systems. The additional consideration of decentralized authorization is geared toward distributed objects. The meta-level access control architecture relies heavily on three concepts: capabilities, message filters and method-based access control. Integrating these features in a meta-object model results in a rich framework for expressing a variety of authorization models for object-oriented systems.

Research efforts in object-oriented systems and in database security have also influenced our work [3, 8, 9, 10, 12, 17, 29, 31]. The ORION/ITASCA system adopts a model of discretionary access control for objects that embraces notions of explicit/implicit, positive/negative and weak/strong authorizations [24]. The model is based on four fundamental access types and also incorporates roles. An extension to this model described in [5] supports additional access types and the modeling of type dependencies. The extended model also clarifies the semantics of subject groups and considers object versions and the potential for distributed authorization control. Type definitions in our model are easily extended to positive/negative and weak/strong authorizations; this would require modifying components to handle the new types. An important advantage of our model is that semantic-based forms of implicit authorization emerge in any object-oriented system designed using MOM.

Fernandez introduced method-based access control for object-oriented systems [15]. By using methods as a basis for access control, first-order access types (e.g., read and write) are reduced to a single execute type. We have chosen method-based access control because it complements MOM nicely. MOM stipulates that access always occurs through a method invocation; for example, reads and writes are implemented using specialized accessor methods. This facilitates policy specifications that are emergent from meta-level access control primitives and MOM.

Multipolicy access control is becoming an important area of research [2, 4, 18]. The mechanisms and models provided by multipolicy systems enable users to protect each object according to a different policy. The architecture described in [4] employs flexible

access control mechanisms and mediators [32]. Mediators are used to shape the access control mechanism so that it can enforce user-specified access control policies. Our work in the area of multipolicy systems seeks a common ground for access control mechanisms that can support the interoperation of disparate authorization policies.

The Argos access control system [20] also shares a goal with the work presented here - that of developing a unified view of heterogeneous access control models in open distributed environments. Argos achieves this goal by incorporating features of various identity-based authorization models. Specifically, it models implicit authorization flow and introduces domains that are used to generate classes of behavior and/or protection rights. These features exploit the semantic richness of the object-oriented paradigm to create a flexible authorization model. Our approach is different in that it decomposes object behavior into primitive mechanisms. Using capabilities is also more general - a capability can be an identity, a group, a label, a role, or an unforgeable ticket in an anonymous transaction.

The Distributed Computing Environment (DCE) requires decentralized authorization services to cope with the difficulties inherent in open distributed environments. DCE's authorization service associates access control lists with servers, files and records, specifying legal operations for each user [14, 25]. Principals (subjects) are registered in a database and assigned group and organization membership. A member's name, group and organization information define the member's privilege attributes. The authorization service works in concert with DCE's authentication service. A member's privilege attributes are embedded in a ticket provided by the authentication server at login. An ACL manager resides on each file server authorizing access requests. DCE supports various kinds of authorization models by allowing customization of ACL managers. However, customization is solely the responsibility of the application developer. While DCE does not deal directly with object-oriented and multipolicy access control issues, it is an attempt to provide ubiquitous, yet practical, access control in distributed systems. MOM must achieve this goal if it is to provide a common secure substrate for heterogeneous distributed objects.

6 Conclusions

The meta-level authorization service architecture presented in this paper integrates primitive capability-based access control mechanisms within a meta-object model for maximal support of multiple policies in het-

erogeneous object systems. The meta-object model engages message filters and method-based access control, although access control for objects is also possible. Access control in this model can be ubiquitous, where each object is responsible for its own authorization policy. It can also be lightweight, where objects provide authorization services for entire object domains. Implemented in the Meta-Object Model (MOM), this authorization service architecture provides a common foundation for the secure interoperation of heterogeneous distributed objects.

References

- [1] G.A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.
- [2] D. Bell. Modeling the "multipolicy machine". In *Proceedings of the New Security Paradigms Workshop*, pages 2-9, 1994.
- [3] E. Bertino, S. Jajodia, and P. Samarati. Access control in object-oriented database systems: Some approaches and issues. In N. Adam and B. Bhargava, editors, *Advanced Database Concepts and Research Issues*, pages 17-44. LNCS 759, Springer - Verlag, 1993.
- [4] E. Bertino, S. Jajodia, and P. Samarati. Supporting multiple access control policies in databases systems. In *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, pages 94-109, May 1996.
- [5] E. Bertino, F. Origgi, and P. Samarati. A new authorization model for object-oriented databases. In J. Biskup, M. Morgenstern, and C. Landwehr, editors, *Database Security VIII, Status and Prospects: Proceedings of the Eighth IFIP WG 11.3 Workshop on Database Security*, pages 199-222, 1994.
- [6] H.H. Bruggemann. Rights in an object-oriented environment. In C. Landwehr and S. Jajodia, editors, *Database Security V, Status and Prospects: Proceedings of the Fifth IFIP WG 11.3 Workshop on Database Security*, pages 99-115, 1992.
- [7] A.A. Chien. *Concurrent Aggregates*. MIT Press, Cambridge, Massachusetts, 1993.
- [8] F. Cuppens and A. Gabillon. A logical approach to model a multilevel object oriented database. In *Proceedings of the Tenth IFIP WG 11.3 Workshop on Database Security*, pages 116-139, July 1996.

- [9] S. Demurjian, T. Daggett, T. Ting, and M. Hu. URBS enforcement mechanisms for object-oriented systems. In D. Spooner, S. Demurjian, and J. Dobson, editors, *Database Security IX, Status and Prospects: Proceedings of the Ninth IFIP WG 11.3 Workshop on Database Security*, pages 79–94, July 1995.
- [10] K. Dittrich, M. Hartig, and H. Pfefferle. Discretionary access control in structurally object-oriented database systems. In C. Landwehr, editor, *Database Security II, Status and Prospects: Proceedings of the Second IFIP WG 11.3 Workshop on Database Security*, pages 105–121, July 1989.
- [11] R. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, 1974.
- [12] E.B. Fernandez, R.B. France, and D. Wei. User group structures in object-oriented databases. In *Database Security VIII, Status and Prospects: Proceedings of the Ninth IFIP WG 11.3 Workshop on Database Security*, pages 57–76, August 1994.
- [13] E.B. Fernandez, R.B. France, and D. Wei. A formal specification of an authorization model for object-oriented databases. In D. Spooner, S. Demurjian, and J. Dobson, editors, *Database Security IX, Status and Prospects: Proceedings of the Ninth IFIP WG 11.3 Workshop on Database Security*, pages 95–110, July 1995.
- [14] Open Systems Foundation. Security in a distributed environment. Technical Report OSF-OWP11-1090-3, Open Systems Foundation, Cambridge, Massachusetts, 1992.
- [15] N. Gal-Oz, E. Guhed, and E.B. Fernandez. A model of methods access authorization in object-oriented databases. In *Proceedings of the Nineteenth International Conference on Very Large Databases*, pages 52–61, July 1993.
- [16] J. Hale, J. Threet, and S. Sheno. A framework for high assurance security of distributed objects. In *Proceedings of the Tenth IFIP WG 11.3 Workshop on Database Security*, pages 76–91, July 1996.
- [17] W. Herndon. An interpretation of clark-wilson for object-oriented database management systems. In T. Keefe and C. Landwehr, editors, *Database Security VII, Status and Prospects: Proceedings of the Seventh IFIP WG 11.3 Workshop on Database Security*, pages 65–85, 1994.
- [18] H. Hosmer. Multipolicy paradigm II. In *Proceedings of the New Security Paradigms Workshop*, 1992.
- [19] S. Jajodia and B. Kogan. Integrating an object-oriented data model with multilevel security. In *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, pages 76–85, May 1990.
- [20] D. Jonscher and K.R. Dittrich. Argos – a configurable access control system for interoperable environment. In D. Spooner, S. Demurjian, and J. Dobson, editors, *Database Security IX, Status and Prospects: Proceedings of the Ninth IFIP WG 11.3 Workshop on Database Security*, pages 43–60, July 1995.
- [21] P. Karger. An augmented capability architecture to support lattice security. In *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, pages 2–12, May 1984.
- [22] P. Karger. Implementing commercial data integrity with secure capabilities. In *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, pages 130–139, May 1988.
- [23] T.F. Keefe, W.T. Tsai, and M.B. Thuraishingham. Soda: A secure object-oriented database system. *Computers & Security*, 8(6):517–533, 1989.
- [24] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–1331, March 1991.
- [25] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly and Associates, Inc., Sebastopol, California, 1993.
- [26] A. Rosenthal, J. Williams, W. Herndon, and B. Thuraishingham. A fine grained access control model for object-oriented dbms. In J. Biskup, M. Morgenstern, and C. Landwehr, editors, *Database Security VIII, Status and Prospects: Proceedings of the Eighth IFIP WG 11.3 Workshop on Database Security*, pages 319–334, 1994.
- [27] M. Schaefer, P. Martel, T. Kanawati, and V. Lyons. Multilevel data model for the trustedontos prototype. In D. Spooner, S. Demurjian, and J. Dobson, editors, *Database Security IX,*

Status and Prospects: Proceedings of the Ninth IFIP WG 11.3 Workshop on Database Security, pages 117-126, July 1995.

- [28] M. Stefik and G. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, December 1985.
- [29] R.K. Thomas and R. Sandhu. Discretionary access control in object-oriented databases: issues and research directions. In *Proceedings of the Sixteenth National Computer Security Conference*, pages 63-74, September 1993.
- [30] J. Threet, J. Hale, and S. Sheno. A process calculus for distributed objects. Technical Report UTULSA-MCS-96-7, The University of Tulsa, Tulsa, Oklahoma, April 1996.
- [31] M.B. Thuraisingham. Mandatory security in object-oriented database systems. *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, 24(10):203-210, October 1989.
- [32] G. Wiederhold. Mediators in the architecture of future information systems: A new approach. *IEEE Computer*, 25(3):38-49, 1992.

Panel II

CORBA Security
Moderator: Catherine McCollum

Work Flow
Chair: John McDermott

An Execution Model for Multilevel Secure Workflows

Vijayalakshmi Atluri

Wei-Kuang Huang

Elisa Bertino

MS/CIS Department
Rutgers University

180 University Avenue, Newark NJ 07102
atluri,waynexh@andromeda.rutgers.edu

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
Via Comelico, 39/41 20135 Milano, Italy
bertino@dsi.unimi.it

Abstract

Workflow management systems (WFMS) support the modeling and coordinated execution of processes within an organization. To coordinate the execution of the various activities (or tasks) in a workflow, task dependencies are specified among them. As advances in workflow management take place, they are also required to support security. In a multilevel secure (MLS) workflow, tasks may belong to different security levels. Ensuring the task dependencies from the tasks at higher security levels to those at lower security level (high-to-low dependencies) may compromise security. In this paper, we consider such MLS workflows and show how they can be executed in a secure and correct manner. Our approach is based on semantic classification of the task dependencies that examines the source of the task dependencies. We classify the high-to-low dependencies in several ways: conflicting vs conflict-free, result-independent vs result-dependent, strong vs weak, and abortive vs non-abortive. We propose algorithms to automatically redesign the workflow and demonstrate that only a small subset among all the types of high-to-low dependencies requires to be executed by trusted subjects and all other types can be executed without compromising security.

The solutions proposed in this paper are directly applicable to another relevant area of research — execution of multilevel transactions in multilevel secure databases since the atomicity requirements and other semantic requirements can be modeled as a workflow. When compared to the research in this area, our work (1) is more general in the sense that it can model several other types of dependencies thereby allowing one to specify relaxed atomicity requirements and (2) is capable of automatically redesigning a workflow without requiring any human intervention by eliminating some cycles among task dependencies that helps to attain higher degree of atomicity.

1 Introduction

Workflow management is a new and emerging area of research. Workflow management systems (WFMS) support the modeling and coordinated execution of pro-

cesses within an organization. WFMS represent today an important, inter-disciplinary area which is commercially very significant, as witnessed by the large number of available products and by the standardization effort undertaken by the Workflow Management Coalition organization. The reason why WFMS are becoming increasingly important is because from an enterprise point of view the effective management of business processes is becoming increasingly crucial. Business processes control which piece of work (task) will be performed by whom and which resources are required and used to accomplish this task. Therefore, a business process specifies how a certain organization will achieve its goals. Optimizing such a process is crucial in today's competitive world. Very often, the use of WFMS is connected with business process reengineering by which business processes are redesigned to achieve significant improvements in critical factors such as cost, quality, service and speed. Several applications are already supported by WFMS, including insurance policy/claims processing, travel expense approvals, healthcare management, system monitoring and exception handling, just to name a few.

To coordinate the execution of the various activities (or tasks) in a workflow, a set of constraints called the task dependencies are specified among them. Task dependencies represent a key component in ensuring the flexibility required to support exceptions, alternatives, compensations and so on, which all arise in real-life activities. An example of constraint is to specify that a certain task must be aborted if another task is aborted. Such a constraint models the fact that if the latter task is not successfully completed, the former task is useless and therefore must be aborted. The development of flexible and powerful WFMS entails many important issues. These systems are thus continuously evolving in order to better satisfy application requirements. In particular, as advances in WFMS take place and their application scope widens, they are also required to support security, meaning that coordination among processes at different security levels has to be supported, indeed without violating security.

In order to ensure correctness and reliability, workflows are associated with a *workflow transaction model* [9]. It is important to note that workflow transaction models must somehow be based on more "flexible" correctness criteria than traditional transaction models. For example, the classical "all-or-nothing" property of the traditional transaction models is not appropriate for workflow transactions. Such a workflow transaction may need to commit some of its actions, while aborting other actions. To satisfy such flexibility requirements, a large number of workflow transaction models have been proposed. Despite the flurry of research and development work around workflow transaction models, security for such transaction models has not been addressed yet. In a multilevel secure workflow transaction (MLS workflow in short), tasks may belong to different security levels. Thus ensuring all the task dependencies, especially those from a task at a higher security level to that at a lower security level, may compromise security. It is easy to understand that in a multilevel environment it is not possible to force the abort of a lower level task upon the abort of a higher level task. The goal of the work we present here is to consider MLS workflows and show how they can be executed in a secure and correct manner.

Our approach begins with a semantic classification of the task dependencies which is based on a close examination of the source of the task dependencies. We argue that only certain types of dependencies can occur in MLS workflows. Then we propose algorithms to automatically (without human intervention) redesign the workflow in such a way that it can be executed in a secure and correct manner. In particular, our approach focuses on redesigning dependencies from higher level tasks to those at lower level because they are the cause for a potential covert channel.

The remainder of the paper is organized as follows. Section 2 reviews the workflow and security models and develops the necessary definitions to formalize our approach. Section 3 presents the multilevel secure workflow model. Section 4 provides an approach to identify the various types of task dependencies based on the semantics of the dependencies. Section 5 shows how all the types of dependencies can be enforced without compromising security. Finally section 6 provides conclusions. Proofs of the theorems are presented in the appendix.

2 The Model

In this section we introduce the basic elements of our workflow model and we summarize the security model we assume.

2.1 The Workflow Model

A workflow is a set of tasks with task dependencies defined among them. A task in its simplest form consists of a set of data operations and task primitives {begin, abort, commit}. Execution of a task, in addition to invoking operations on data items (either read or write),

requires invocation of these task primitives. All data operations in a task must be executed only after the begin primitive is issued. All tasks must end with either a commit or abort.

A primitive may move a task from one state to another. A task (t_i) can be in one of the following states: initial state (in_i), execution state (ex_i), commit state (cm_i) or abort state (ab_i). (We use b_i , a_i and c_i to denote the begin, abort and commit primitives of t_i .) For instance, a task may move from its initial state to the execution state by invoking the begin primitive.

To control the coordination among different tasks, dependencies are specified based on these task primitives. Task dependencies in turn can either be *static* or *dynamic* in nature. In the static case, the workflow is defined well in advance to its actual execution, whereas dynamic dependencies develop as the workflow progresses through its execution [14]. Task dependencies may exist among tasks within a workflow (*intra-workflow*) or between two different workflows (*inter-workflow*). In [13], three basic types of task dependencies have been identified: *control-flow dependencies*, *value dependencies* and *external dependencies*. Control-flow dependencies may in turn involve explicit transmission of data as part of the result of a task. We call such dependencies *control-flow dependencies with data flow*. In the remainder of this section, we briefly discuss the four types of dependencies and introduce some basic definitions concerning workflows.

2.1.1 Control-flow Dependencies

A control-flow dependency can be defined as follows: A task t_j can enter state st_j only after task t_i enters state st_i . Control-flow dependencies can be modeled based on the ACTA framework [6].

Given two extended transactions t_i and t_j , a list of possible control-flow dependencies is presented below.

1. Strong Commit Dependency: If a task t_i commits then t_j must commit (represented as $t_i \xrightarrow{ac} t_j$).
2. Abort Dependency: A task t_j must abort if t_i aborts (represented as $t_i \xrightarrow{a} t_j$).
3. Termination Dependency: A task t_j can terminate (either commit or abort) only after the completion (commit or abort) of t_i (represented as $t_i \xrightarrow{t} t_j$).
4. Begin Dependency: A task t_j cannot begin until t_i has begun (represented as $t_i \xrightarrow{b} t_j$).
5. Begin-on-Commit Dependency: A task t_j cannot begin until t_i commits (represented as $t_i \xrightarrow{bc} t_j$).
6. Force Begin-on-abort Dependency: A task t_j must begin if t_i aborts (represented as $t_i \xrightarrow{fba} t_j$).
7. Exclusion: Given any two tasks t_i and t_j , if t_i commits t_j must abort, or vice versa (represented as $t_i \xrightarrow{e} t_j$).
8. Weak begin-on-commit: Given any two tasks t_i and t_j , t_j can begin if t_i commits, (represented as $t_i \xrightarrow{wbc} t_j$).

9. Group Commit: Given any two tasks t_i and t_j , either both t_i and t_j commit or neither commits [4]* (represented as $t_i \xrightarrow{gc} t_j$ or $t_j \xrightarrow{gc} t_i$).

A comprehensive list of task dependencies based on the three task primitives, namely, begin, commit and abort, can be found in [8, 6], which include commit, weak-abort, force-commit-on-abort, serial, begin-on-abort and weak-begin-on-commit dependencies. In general, according to the logical nature of dependency, they can be either *strong* or *weak*.

Definition 1 Given a control-flow dependency $t_i \xrightarrow{x} t_j$, if the dependency implies a logical relationship $st_i \Rightarrow st_j$ ($st_i \Leftarrow st_j$), we say that it is *weak* (*strong*). \square

According to the above definition, a *strong* dependency specifies a logical relationship such that t_j can enter state st_j only if task t_i enters state st_i (e.g., bc, b, t, gc etc.). On the other hand, a *weak* dependency states that if t_i enters state st_i , then t_j can/must enter state st_j , but t_j can enter st_j even t_i has not entered st_i (e.g., sc, a, fba, wba, wbc, c, e, etc.). That is, the semantic difference between the *strong* and *weak* dependencies is that the former specifies the necessary condition for t_j to enter st_j whereas the latter the sufficient condition.

Definition 2 Given a control-flow dependency $t_i \xrightarrow{x} t_j$, the dependency is of type *abortive* if st_j is *abj*; otherwise it is *non-abortive*. \square

The classification given by the above definition applies to both strong and weak dependencies. For example, *abortive* type dependencies include abort, exclusion dependency, etc. whereas *non-abortive* type dependencies are commit, strong commit, begin on commit, serial, begin on abort, etc..

No-Force-Commit and No-Prevent-Abort Assumptions:

When applying those dependency definitions to the workflow environment, two tacit assumptions must be made because of the relaxed atomicity requirement.

No-Force-Commit (NFC) Assumption: No task execution can be guaranteed to commit. However, a task can be forced to begin or abort.

No-Prevent-Abort (NPA) Assumption: No task execution can be prevented from aborting. However, a task can be prevented from beginning or committing. Further examination of these dependencies reveals that all abortive type workflow dependencies must be weak, however, non-abortive type workflow dependencies can

be either weak or strong. For instance, some weak non-abortive type dependencies such as sc, force-commit-on-abort (fca) (t_j must commit if t_i aborts), and gc, etc. violate the NFC assumption and all strong abortive type dependencies such as termination dependency (t) violate the NPA assumption.

2.1.2 Control-flow Dependencies with Data-flow

A control-flow dependency with data-flow can be defined as follows: A task t_j can enter state st_j after task t_i enters state st_i and t_i passes values of data objects to t_j . In these dependencies, in addition to the control flow, there could even be information flow (or data flow) between the tasks where a task needs to wait for data from another task. Notice that control-flow dependency with data-flow is meaningful only for limited combinations of st_i and st_j . For example, st_i and st_j can be "commit" and "begin," respectively, but cannot be "begin" and "commit."

2.1.3 Value Dependencies

A value dependency can be defined as follows: A task t_j can enter state st_j only after task t_i 's outcome satisfies a condition c_i . The condition in the above statement can be a logical expression whose value is either 0 or 1. Note that this dependency is different from the control-flow dependency with the data flow. For example, " t_j can begin if t_i is a success (semantically)."[†]

2.1.4 External Dependencies

Unlike the prior two types, external dependencies are caused by some parameters external to the system, such as time. An external dependency can be defined as follows: A task t_i can enter state st_i only after if a certain condition c_j is satisfied where the parameters in c_j are external to the workflow. Examples include a task t_i can start its execution only at 9:00am or task t_j can start execution only 24hrs after the completion of task t_k .

2.1.5 Definitions

In the following, we develop the necessary definitions for formalizing the execution model for MLS workflows.

Definition 3 A workflow W can be defined as a directed graph whose nodes are the tasks $t_1, t_2 \dots t_n$ in the workflow and edges are the task dependencies $t_i \xrightarrow{x} t_j$, where $t_i, t_j \in W$ and x denotes the type of dependency. \square

Definition 4 Two operations $o_i[d]$ and $o_j[d]$ conflict with each other if they operate on the same data object d and at least one of them is a write. \square

Given a dependency $t_i \xrightarrow{x} t_j$, we say t_i is the *parent* of t_j and t_j the *child* of t_i .

Definition 5 Given two tasks t_i and t_j in W , t_i is said to be an *ancestor* (*descendent*) of t_j , if t_i is a *parent*

*Group commit involving a set of tasks can be defined using pairwise group dependencies.

[†]Failure of a task does not necessarily mean abort of a task. A task may still semantically fail even if it successfully commits.

(child) of t_j or t_i is a parent (child) of t_k where t_k is an ancestor (descendent) of t_j . \square

Definition 6 Given a workflow W , a *potential-state-set* of W (denoted as $PSS(W)$) is a set of vectors such that each vector (called *potential-state*) in $PSS(W)$ represents an allowed combination of final states of all the tasks in W . \square

As an example, consider the workflow $W = \{t_1, t_2\}$, and the dependency $t_1 \xrightarrow{sc} t_2$. $PSS(W) = \{(cm_1, cm_2), (ab_1, ab_2), (ab_1, cm_2)\}$. Note that a different dependency between t_1 and t_2 may result in an entirely different $PSS(W)$.

Definition 7 Given two workflows W and W' , we say that W' covers W if (1) both W and W' consist of the same set of data operations, (2) for every pair of conflicting operations $o_i[d]$ and $o_j[d]$, if t_i is an ancestor of t_j in W , then t_i must be an ancestor of t_j in W' , and (3) for each $P \in PSS(W)$, there exists a $P' \in PSS(W')$ such that $P \subseteq P'$. \square

2.2 The Security Model

We assume the security structure to be a partially ordered set \mathcal{S} of security levels with ordering relation \leq . A class $s_i \in \mathcal{S}$ is said to be *dominated* by another class $s_j \in \mathcal{S}$ if $s_i \leq s_j$. A class s_i is said to be *strictly dominated* by another class s_j (denoted as $s_i < s_j$) if $s_i \leq s_j$ and $i \neq j$.¹

Let D be the set of all data objects. Each data object $d \in D$ is associated with a security level. Every task t_i in a workflow W is associated with a security level. We assume that there is a function L that maps all data objects and tasks to security levels. That is, for every $d \in D$, $L(d) \in \mathcal{S}$, and for every task $t_i \in W$, $L(t_i) \in \mathcal{S}$. We require every task to obey the following two security properties — the simple security and the restricted \star -property.

1. A task t_i is allowed to read a data object d only if $L(d) \leq L(t_i)$
2. A task t_i is allowed to write to a data object d only if $L(d) = L(t_i)$.

In addition to these two restrictions, a *secure* system must prevent illegal information flows via covert channels.

3 Multilevel Secure Workflows

A multilevel secure (MLS) workflow may consist of tasks of different security levels (as in example 1 below). Thus, an MLS workflow consists of nodes at different security levels where the dependency edges may connect tasks of either the same security level or different security

levels, which can be distinguished as follows. The dependency edge connecting tasks of the same security level is referred to as *intra-level dependency* and the one connecting tasks of different security levels as *inter-level dependency*. Since intra-level dependencies by themselves cannot violate any multilevel security constraints and are not different from the task dependencies in a non-secure environment, hereafter we concentrate only on inter-level dependencies. We further divide inter-level dependencies into two categories: *high-to-low*² and *low-to-high* since their treatment has to be different in a MLS environment because of its “no downward information flow” requirement.

Example 1 Consider a workflow that computes the weekly pay of all employees at the end of each week. This process involves several tasks as follows. Task t_1 : compute the number of hours worked by an employee h , which is the sum of regular hours worked (n) and overtime hours worked (o) by the employee during that week, Task t_2 : calculate the weekly pay of an employee (p) by multiplying h with the hourly rate of the employee (r), and Task t_3 : after computing the pay for the week, reset h , n and o to zero. The information about hourly rate (r) and weekly pay (p) is considered sensitive, and therefore both r and p are classified *high*, while the rest of the information is classified *low*. Since this workflow involves write operations at different levels, it is a MLS workflow.

According to the two restrictions of our security model, since t_1 and t_3 write objects at *low* (h , n and o) they must be *low* tasks, and since t_2 reads the *high* object (r) and writes the *high* object (p), it must be a *high* task.

Moreover, the following task dependencies exist: task t_2 can begin only after t_1 commits, thus $t_1 \xrightarrow{bc} t_2$, and t_3 can begin only after t_2 commits, thus $t_2 \xrightarrow{bc} t_3$, as shown in figure 1. While $t_1 \xrightarrow{bc} t_2$ is a *low-to-high* dependency, $t_2 \xrightarrow{bc} t_3$ is *high-to-low*. Thus it is an MLS workflow. \square

Execution of a workflow involves (1) enforcing all task dependencies, (2) assuring correct execution of interleaved workflows, and (3) ensuring that the workflow terminates in one of the predefined acceptable states. In this paper, we focus on the first part only.

It follows from the no covert channel requirement that for any given task dependency $t_i \rightarrow t_j$, $L(t_i) \not< L(t_j)$. That is, to prevent covert channels, no *high-to-low* dependency must be enforced.

In a correct MLS workflow specification, it is not possible to have a high-to-low value dependency because enforcing such dependency amounts to directly sending data from a higher to a lower security level. The same

¹Here we made an assumption that $s_i \neq s_j$ iff $i \neq j$.

²Although we use the term *high-to-low*, this dependency also includes those among two incomparable security levels.

argument applies to the case of a high-to-low control flow dependency with data flow.

With respect to external dependencies, for the purpose of our work, we categorize them as *absolute* and *relative*, where absolute dependencies are solely controlled by the external factors, whereas relative dependencies are specified as external parameters but are controlled by the internal events. For example, “a task t_i can start its execution only at 9:00am,” is an absolute external dependency, whereas “task t_j can start its execution only 24hrs after the completion of task t_k ,” is a relative external dependency. We need this classification because enforcement of these two types is different in MLS environment. While absolute external dependencies can be enforced without compromising security, relative external dependencies may be exploited to establish a covert channel, especially when this dependency is from a *high* task to a *low* task.

A relative external dependency is nothing but a control flow dependency with additional temporal constraints. Since temporal constraints cannot be modeled by simple graph structures but require special modeling techniques that can incorporate external events, we do not consider them in this paper. Therefore, in this paper, we consider only the control-flow dependencies.

3.1 Execution Criteria

In the following, we define four levels of execution based on the degree of security or correctness that it guarantees. First we recall the definition of secure execution from [12]. An execution is said to be secure if it satisfies the non-interference property [10], i.e., no lower level task is effected by any higher level task.

1. **SSSC-level** (strongly-secure and strongly-correct): An MLS workflow execution is said to be of SSSC-level, if it is secure and all the task dependencies are enforced. This calls for complete elimination of covert channels, yet enforcing all dependencies. This is the most desirable case.
2. **SSWC-level** (strongly-secure and weakly-correct): An MLS workflow execution is said to be of SSWC-level, if it is secure but all the task dependencies need not be enforced. This requires complete elimination of all covert channels, however one need not enforce all the task dependencies.
3. **WSSC-level** (weakly-secure and strongly-correct): An MLS workflow execution is said to be of WSSC-level, if it enforces all the task dependencies but may allow a low bandwidth covert channel. The bandwidth of the covert channel can be reduced by introducing noise or introducing a fixed delay.
4. **WSWC-level** (weakly-secure and weakly-correct): An MLS workflow execution is said to be of WSWC-

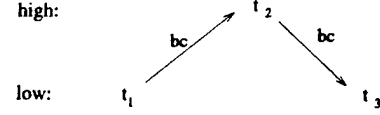


Figure 1: Task dependencies in the MLS workflow in Example 1

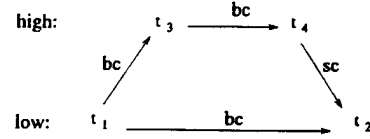


Figure 2: Task dependencies in the MLS workflow in Example 2

level, if it does not enforce all the task dependencies, yet allows covert channels.

Although it is desirable to have the first level of execution, this level is difficult to achieve due to the inherent conflicts between security and correctness. [2] proposes an approach to eliminate all high-to-low task dependencies. (i.e. ensures SSWC-level execution). In this paper, we show how SSSC-level of execution can be attained.

4 Semantic Classification of Task Dependencies in MLS Workflows

In this section, we take a closer look at all types of dependencies and examine what they semantically mean in an MLS environment. We give more insight into each type of dependency in MLS environment and argue that only some types of dependencies can be specified in MLS workflow specification. Note that our arguments focus only on high-to-low dependencies because, as we argue in section 5.1, low-to-high dependencies can be enforced without compromising security requirements.

To reason about the semantics of high-to-low dependencies (control-flow), we first would look at the source of this dependency and categorize them as follows:

1. This first category of dependencies arises to force the order of (conflicting) operations on shared data objects.
2. The second category of dependencies arises to force properties such as atomicity, mutual exclusion, etc.

Consider the task dependency $t_2 \xrightarrow{bc} t_3$ in example 1. The intention of this dependency is to avoid overwriting of n and o by t_3 before t_2 reads them. Thus, the source of this dependency is to force a specific order on the conflicting operations, and therefore belongs to the first category. Dependencies in the second category are specified according to the semantics of the workflow.

For example, the semantics require that either one of two tasks must commit but not both (mutual exclusion). For instance, consider a travel reservation workflow, where two tasks are purchasing a ticket in Delta and in United, where only one task must commit but not both. In the following, we provide an example illustrating such dependencies in an MLS environment.

Example 2 Consider a workflow that arranges a travel schedule for a person P. Assume that P has to first make a trip from Washington D.C. to Toronto and then from Toronto to Moscow. The second part of the trip is on a secret mission and therefore has to be considered as highly sensitive information and thus assumes high level. However, the first part of the trip is not classified and thus is considered low. Assume this workflow consists of the following four tasks: reserving a ticket for the first part of the trip (denoted as t_1), purchasing the ticket for the first part (t_2), reserving a ticket for the second part of the trip (t_3), and purchasing the ticket for the second part of the trip (t_4), where t_1 and t_2 are low level tasks and t_3 and t_4 are high level tasks. The following task dependencies exist: Purchasing a ticket cannot be started unless reserving the ticket is complete. Thus, $t_1 \xrightarrow{bc} t_2$ and $t_3 \xrightarrow{bc} t_4$. Moreover, reserving a flight for the second part of the trip has to be done only after making sure that the flight is available for the low part of the trip, thus $t_1 \xrightarrow{bc} t_3$. Furthermore, purchasing the ticket for the low part of the trip must be committed only if purchasing the ticket for the high part of the trip is successful, thus, $t_4 \xrightarrow{sc} t_2$. While the first two task dependencies are intra-level dependencies, the latter two are low-to-high and high-to-low dependencies, respectively, as shown in figure 2. \square

The intention of the high-to-low dependency $t_4 \xrightarrow{sc} t_2$ in the above example is to capture the semantics of the workflow rather than forcing an order between conflicting operations. Thus this dependency belongs to the second category.

If we examine once again the two types of the source of dependencies and analyze their effect on the workflow, we can make the following observation. Imagine the following two scenarios: in the first, assume the high-to-low dependency $t_i \rightarrow t_j$ is enforced, whereas in the second, this dependency is not enforced. With the second category of dependency (e.g., $t_4 \xrightarrow{sc} t_2$ in example 2) the result of t_j might be different if $t_i \rightarrow t_j$ is not enforced than from the case when it is enforced. In other words, the result of t_j might be affected if $t_i \rightarrow t_j$ is not enforced. However, the dependency $t_i \rightarrow t_j$ does not impact the result of t_i . Thus we call the second category of dependencies *result-dependent* (RD). On the other hand, consider the first category of dependencies (e.g., $t_2 \xrightarrow{bc} t_3$ in example 1). If the dependency is not enforced, the result of t_j

will not be affected but that of t_i will be affected. This can only occur when the two tasks share common data in multilevel secure systems (when this dependency is high-to-low). Thus, we call the first category of dependencies *conflicting* (CN). In the following, we formally define these categories.

Definition 8 A dependency between two tasks $t_i \xrightarrow{x} t_j$ is said to be *conflicting* (CN) if there exist at least two conflicting operations $o_i[d]$ and $o_j[d]$ ($i \neq j$); otherwise $t_i \xrightarrow{x} t_j$ is said to be *conflict-free* (CF). \square

On the other hand, from the perspective of task result, a dependency can either be *result-independent* or *result-dependent*. Formally:

Definition 9 A dependency $t_i \xrightarrow{x} t_j$ is said to be *result-dependent* (RD) if the result of executing the child is different when the dependency is enforced from that when it is not enforced; otherwise $t_i \xrightarrow{x} t_j$ is said to be *result-independent* (RI). \square

The intuitive idea behind this classification is that the result of the execution of either the child (in case of RD) or the parent (in case of CN) must be different when the dependency is enforced than from the case when it is not enforced. Thus, there cannot be any dependency which is both conflict-free and result-independent.

Let us now examine this categorization in the wake of multiple security levels on data items and tasks. At this point, our primary concern is how to enforce high-to-low dependencies in a MLS workflow.

- Case CN: Dependencies belonging to this category indicate that the two tasks involved in the dependency access some common data items in conflicting mode. The primary reason to enforce this type of dependency is to enforce the order of these conflicting operations. This category depicts a typical conflicting situation where the parent with a read operation is followed by the child with a write operation on the same data object (No other combination of read-writes are possible as per our security model). High-to-low CN dependencies can be further classified into result-dependent (RD) and result-independent (RI) dependencies, thus resulting in CN-RD dependencies and CN-RI dependencies. These two categories of dependencies are briefly discussed below.

- CN-RI: RI dependencies mean that the result of the child does not depend on whether the dependency is enforced or not. However, no enforcement of this dependency may produce a different result for the parent task. Obviously the result of the child is independent of whether

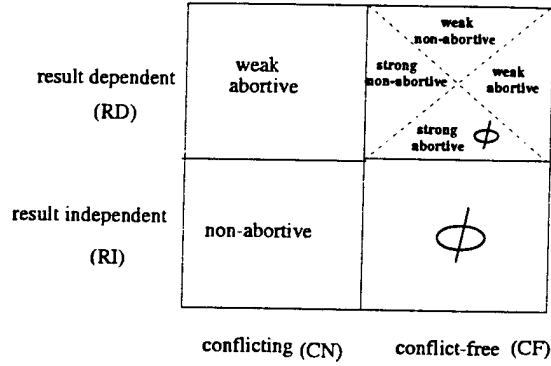


Figure 3: Categorization of high-to-low dependencies

the dependency is enforced or not. Therefore, all dependencies falling into this category should be non-abortive such as begin on commit, begin on abort, etc. because otherwise the result of the child will get affected (i.e., abort) by the parent.

- CN-RD: Dependencies such as force abort, termination, exclusion etc. that may cause the child task to get aborted (the *abortive* type) will fall under RD category. *Abortive* type dependencies are all RD because without enforcing the dependency, the child task may commit (as opposed to abort). This is because all *abortive dependencies* could possibly cause the abortion of the child and therefore are not of type RI.
- Case CF: Dependencies belonging to this category can only be formed by pure semantic specification. Although two tasks do not conflict, sometimes for ensuring an acceptable termination state of the workflow, such dependencies are specified.
 - CF-RD: These dependencies are specified in such a way that the execution of the child depends on the value or outcome of the parent. Therefore altering the execution order between these two tasks would affect the outcome of the child, in other words, the dependency is result-dependent. These dependencies can be either strong or weak.
 - CF-RI: A CF type dependency does not cause any effect on the result of the parent task. Thus, there cannot exist any dependency which is both CF and RI because its presence neither effects the parent nor the child.

The Venn-diagram shown in figure 3 depicts these various categories of high-to-low dependencies. The above categorization of high-to-low dependencies is important, because each category needs to be handled according to

a different approach in an MLS environment. The specific approach used for each category will be presented in the next section.

We introduce now an algorithm to classify the high-to-low dependencies in a given workflow according to the above classification. The algorithm only needs to know the set of data that will be potentially read and written by each task and the set of dependencies among tasks. Note, however, the approach we introduce in the next section still applies even if a task only reads (writes) a subset of such set.

Algorithm 1 [Identifying the Type of Dependency]

```

for every  $t_i \xrightarrow{x} t_j$  in  $W$  where  $L(t_j) < L(t_i)$ ,
/* for every high-to-low dependency */
  if  $\exists r_i[d] \in t_i$  and  $w_j[d] \in t_j$  where  $i \neq j$ 
    /* if two tasks are conflicting */
    if  $t_i \xrightarrow{x} t_j$  is abortive
      label  $x$  with CN-RD
      /* abortive CN type dependencies are RD */
    else
      label  $x$  with CN-RI
      /* non-abortive CN type dependencies are RI */
  else
    label  $x$  with CF-RD
    /* all conflicting free dependencies must be RD */
end{for} □

```

5 Execution of MLS Workflows

Enforcing a *low-to-high* dependency will not result in violation of security. By contrast, a covert channel may be established while enforcing a *high-to-low* dependency. The *high-to-low* dependencies are, therefore, much more difficult to handle than *low-to-high* dependencies. In the remainder of this section, we first briefly summarize two possible approaches to enforcing *low-to-high* dependencies. We then discuss approaches to enforcing *high-to-low* dependencies which are the focus of our paper.

5.1 Low-to-High Dependencies

Consider a task dependency $t_i \xrightarrow{bc} t_j$ such that $L(t_i) < L(t_j)$, meaning that t_j can begin only after t_i 's commit. Enforcing such a dependency requires the use of a mechanism by which the higher-level task t_j is activated upon commit of t_i . Several approaches can be devised.

A first approach is based on the use of triggers. Under this approach, a trigger would be incorporated into t_i . Thus t_i activates a trigger upon its commit at the *high* level, at which point the *high* task t_j can begin. This will not violate security since it is equivalent to writing-up. To ensure that the trigger is delivered from low to high, and then to increase reliability, this approach can be complemented by mechanisms supporting reliable transfer of messages in multi-level systems. Recently, an approach called PUMP has been proposed

[11] which provides a reliable transfer of messages from lower to higher levels with a controlled stream of acknowledgments from higher to lower levels. The PUMP mechanism, however, may be exploited by malicious programs as covert channels, even though with a very small bandwidth. An analysis has been carried out in [11] to measure the bandwidth of the covert channel.

Another approach is based on testing a given precondition. Such precondition has to be satisfied to begin the high task t_j . This can be implemented by making the high task to read some data at low level and check for the satisfaction of the precondition periodically. Note that this approach does not require a secure message passing as in the earlier approach. On the other hand, it requires the high task to poll some low data to test for the precondition.

5.2 High-to-Low Dependencies

In the following section, we present our approach to handle high-to-low dependencies. Since CN-RI dependencies are conflicting, the main issue is how to synchronize the tasks to satisfy the dependency without introducing covert channels. Our approach to handle CN-RI dependencies eliminates the high-to-low dependency by *splitting* the high level task. The purpose of a CF-RD dependency is to force a low level task to move to a certain state according to the state of a high level task. Our approach to handle CF-RD compensates the low task when necessary by executing an inverse dependency. Finally, we view CN-RD as a combination of CN-RI and CF-RD since it could be due to the conflicting operations as well as due to the semantics of the workflow.

5.2.1 Enforcing CN-RI type high-to-low Dependencies

As described in the earlier section, these dependencies arise if there exists a high task that must read a low data item before it is modified by another low task. As in example 1, the intention of the dependency $t_2 \xrightarrow{bc} t_3$ is to prevent t_3 from overwriting the low values of data items n and o yet to be read by t_2 . Since delaying t_3 until t_2 's commit would result in a covert channel, we propose two possible approaches to tackle this problem. The first approach is based on the multiple versions approach, presented in [7]; the second approach is new and proposed by this paper.

Maintaining Multiple Versions: One may use multiple versions of data to cope with such high-to-low dependencies. Whenever a task writes a data item d , a new version of d is created, thus the value yet to be read by a high transaction is reserved as an older version of d . Costich and Jajodia [7] have proposed an approach in which they associate an index with each read/write operation. When a multilevel transaction first updates d , it is indexed by 1, the next write to d is indexed by 2, and so on, and when it reads d , the read operation is assigned

the same index as that of the previous write operation. Thus, this indexing is used to preserve the dependencies by allowing a high task (which they call section of the multilevel transaction) to read an appropriate version of d in order to enforce the dependency. This approach has the major drawback of requiring a special-purpose multiversioning concurrency control mechanism. The approach we propose here (described below) does not have such requirement and therefore can be supported by any standard DBMS. A similar approach to preserve the high-to-low dependency has been proposed by Smith et al. [15]. This uses a cache to save data to be read by a high section so that even if a low section overwrites the data yet to be read by the high section, the dependency is still preserved. Our approach of splitting the task (presented below) provides a framework to enforce the high-to-low dependency which can be implemented using the Smith et al.'s caching scheme.

Splitting the High Task: According to our approach, first all the operations in every task are reordered in such a way that all read operations on lower level data items occur before all operations on data items at the level of the task (Since all these are read operations, it does not affect the correctness of the task.). Then the task is divided into partitions based on the data items it is accessing. For example, if a high task t_2 in example 1 is split into two tasks, the first task t_2^{low} contains all the low read operations, and the second task t_2 all the high read/write operations. We introduce a begin-on-commit dependency with data flow $t_2^{low} \xrightarrow{bc} t_2$ to ensure that data read by the read operations in t_2^{low} in fact are carried over to t_2 even in the wake of other interfering tasks. Then, to enforce $t_1 \xrightarrow{bc} t_2$ and $t_2 \xrightarrow{bc} t_3$, we convert these two dependencies as $t_1 \xrightarrow{bc} t_2^{low}$ and $t_2^{low} \xrightarrow{bc} t_3$, as shown in figure 5. The low task t_3 proceeds only if t_2^{low} commits, thus preserving the high-to-low dependency between t_2 and t_3 . In the following, we formally present our approach.

Definition 10 Given a task t_i , given $s < L(t_i)$, we say that there exists a partition t_i^s of t_i if $t_i^s = \{o_i[d] \in t_i | L(d) = s\} \neq \emptyset$. \square

Since a task t_i is allowed to read and write data at its own level as well as read data from lower levels, all read operations pertaining to a lower level s belong to one partition (say t_i^s), whereas all read and write operations pertaining to the level of the task belong to another partition (we use the original identity t_i). Thus if a task reads from two lower levels and reads or writes at its own level, it has three partitions.

Definition 11 Given two tasks t_i and t_j in W , t_j is said to be a *closest-s-ancestor* of t_i if (1) $L(t_j) = s$, (2) t_j is

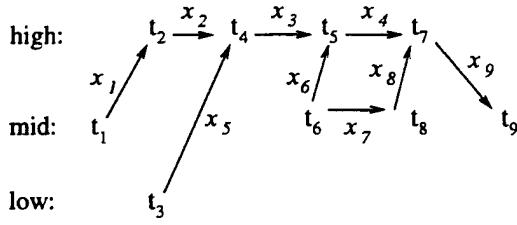


Figure 4: An example demonstrating the closest ancestor high:

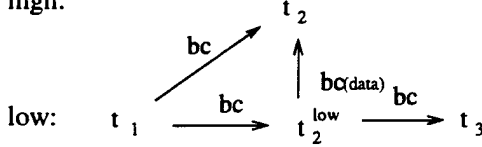


Figure 5: Modified Workflow after executing *split_task* for Example 1

an ancestor of t_i , and (3) there exists no t_k in W such that $L(t_k) = s$ and t_k is an ancestor of t_i and descendant of t_j . \square

For example, in the workflow shown in figure 4, t_8 and t_1 are the closest-mid-ancestors of t_7 and t_3 is the closest-low-ancestor of t_7 . By contrast, t_6 is not a closest-mid-ancestor of t_7 because there exists t_8 which is an ancestor of t_7 and a descendant of t_8 .

The following algorithm specifies our approach to task splitting.

Algorithm 2 [*split_task*]

```

for every  $t_i \xrightarrow{x_{dep}} t_j$  in  $W$  where  $dep$  is CN-RI
  split  $t_i$  into number of partitions where each partition
   $t_i^s = \{o_i[x] \in t_i \mid L(x) = s \wedge s < L(t_i)\}$ 
  remove every operation  $o_i[d]$  from  $t_i$  such that
   $L(d) < L(t_i)$ 
  /* divide the high parent such that each partition
  consists of operations involving access to data items
  of the same security level */
  add all  $t_i^s$  to  $W$ 
  for each  $t_i^s$  such that  $s < L(t_i)$ 
    add an edge  $t_i^s \xrightarrow{bc(data)} t_i$ 
    /*add a bc dependency with data-flow from
    every lower level partition to that at level  $(t_i)^*$ */
    for every  $t_k$  in  $W$  where  $t_k$  is the closest-s-ancestor
    of  $t_i$ ,
      add an edge  $t_k \xrightarrow{bc} t_i^s$ 
    end{for};
  end{for};
  add an edge  $t_i^s \xrightarrow{x} t_j$  such that  $s = L(t_j)$ 
end{for}  $\square$ 

```

According to the above algorithm, given a dependency $t_i \xrightarrow{x_{CN-RI}} t_j$, we first split t_i into number of partitions

based on the security level of data items involved in each operation of t_i . Then we add an edge of type "bc" (i.e., begin-on-commit) with data flow from every lower level partition (t_i^s) to the highest level partition (t_i). This is to ensure that the data read by a low partition reaches the high partition. Later, for each level where there is a partition of t_i , we add an edge of type "bc" from every closest ancestor t_k of t_i at that level to that partition. This edge is to ensure that splitting does not remove any dependency from t_k to t_i^s . We need to take care of the case where t_k and t_i^s are conflicting. The abortive type need not be enforced because we need not to preserve the order of conflicting operations if t_i^s is to abort because t_i^s is always read-only. Thus $t_k \xrightarrow{x} t_i^s$ is always of type "bc."

However, in the above algorithm, we do not add an edge from a closest-s-ancestor t_k of t_i to a partition $t_i^{s'}$ where $s' < s$ because the dependency path from t_k to t_i is not meant to capture the dependency from a lower level s' . For example, in the workflow shown in figure 4, if t_7^{mid} exists and t_7^{low} does not exist, we need to add dependencies $t_1 \xrightarrow{bc} t_7^{mid}$ and $t_8 \xrightarrow{bc} t_7^{mid}$ but do not need to add any dependency from t_3 though it is a closest-ancestor of t_7 . On the other hand, if t_7^{mid} does not exist but only t_7^{low} exists, then we need to add only the dependency $t_3 \xrightarrow{bc} t_7^{low}$. In the last step, we add an edge from the partition at the level of t_j to t_j . This is of the same type of the original dependency.

Theorem 1 Let W be a workflow. Let W' be the workflow obtained from W by applying algorithm 2 to W . Then, W' covers W . \square

5.2.2 Enforcing CF-RD type high-to-low Dependencies

As noted earlier, weak abortive, strong non-abortive and weak non-abortive dependencies fall into the CF-RD category. We discuss them in separate sections as the first two require a different treatment from the last one.

Weak abortive and strong non-abortive type. In this section, we first present a straightforward approach employed in many systems, that is based on the use of a buffer. It has, however, the drawback of introducing some covert channel, even though with a limited bandwidth. Then we present our approach, based on using compensating tasks, which does not have such drawback. According to our approach, sometimes the compensating tasks need to be executed by a trusted subject such as a human user. Note that this is not necessarily a drawback because workflow systems are designed to support and allow user interactions. Therefore, requiring the intervention of a user is natural in a workflow environment

Using a buffer. An approach is to use a buffer at *high* (assume its size is sufficiently large) in which the commit message of the *high* task is stored. This message will first be subject to a delay of some random duration, and then will be transmitted to *low*. If several such messages of a single workflow get accumulated during the delay period of the first message, these messages cannot be sent at the same time, but must be sent individually with the delay incorporated in between each of them. Thus, though there exists a channel of downward information flow, the bandwidth of this channel would be low. (However, if the bandwidth of the channel does not exceed 100 bits per second, then it is fully secure (at B3 or A1 level).) It is important to note that this approach might affect the performance of the system because the signal from *high* is delayed thereby delaying all the *low* tasks unnecessarily.

Running Compensating Tasks. In this paper, we propose an alternative approach to enforce a RD type of *high-to-low* dependency by executing a compensating task. We show below that while some high-to-low dependencies (strong non-abortive and weak abortive type) can be enforced by compensating a *low* task. Note that the *low* compensating task needs to be initiated by a trusted subject. This approach, however, is applicable only to certain tasks for which there exist a compensating task.

Definition 12 A task t_i is said to be *compensatable* if the effects of its execution can be semantically undone by executing a compensating task t_i^{-1} . \square

Definition 13 If there exists a compensating task t_i^{-1} for a task t_i , then $L(t_i^{-1}) = L(t_i)$. \square

Definition 14 Given a dependency $t_i \xrightarrow{x} t_j$ of type x we define an *inverse* x^{-1} for x such that for each $P \in PSS(W)$, there exists a $P' \in PSS(W')$ such that $P \subseteq P'$, where

if x is either strong non-abortive or weak abortive, $W = \{t_i, t_j\}$ with the dependency $t_i \xrightarrow{x} t_j$ and $W' = \{t_i, t_j, t_j^{-1}\}$ with the dependency $t_i \xrightarrow{x^{-1}} t_j^{-1}$ in which t_j^{-1} is a compensating task of t_j , \square

In order to derive such an inverse for each dependency, we have made the following assumptions.

Given a task t_i and its compensating task t_i^{-1} ,

1. $ex_i \Rightarrow$ either ab_i or cm_i
2. $ex_i^{-1} \Rightarrow cm_i^{-1}$
3. $\sim ex_i \equiv ab_i$
4. $\sim ex_i^{-1} \equiv ex_i$
5. $cm_i^{-1} \equiv \sim ex_i$
6. $\sim cm_i \equiv ab_i$
7. $\sim ab_i \equiv cm_i$
8. $cm_i^{-1} \equiv ab_i$

The first assumption indicates that every task that has started its execution must either commit or abort. The second assumption states that a compensating task must always commit. The third assumption states that a task that has not yet begun is functionally equivalent

x	bc	a	e	ba	serial
x^{-1}	fba	fba	sc	sc	fba

Table 1: Some RD type dependencies and their inverses

to (denoted as \equiv) the state where the task has started its execution but has aborted. Assumption 4 says that if a compensating task has not started its execution, then this state is same as that of proceeding with the execution of the corresponding task. Similarly, assumption 5 states that the successful completion of a compensating task results in the same state when its corresponding task has not started its execution. Assumptions 6 and 7 state that abort and commit are complements of each other. Assumption 8 states that the successful completion of a compensating task is functionally equivalent to undoing all the effects of its corresponding task. The first five are referred to as the basic assumptions while the last three can be derived from the basic assumptions.

Based on the above assumptions, we can derive the inverse dependency for any given dependency. In the following example, we show our reasoning about how $t_i \xrightarrow{bc} t_j$ is equivalent to $t_i \xrightarrow{fba} t_j^{-1}$. Recall from section 2.1.1 that $t_i \xrightarrow{bc} t_j$ represents the dependency that task t_j cannot begin until task t_i commits, which implies the following logical relationship: $cm_i \Leftarrow ex_j$. It results in the following PSS: $\{(cm_j, cm_i), (ab_j, cm_i), (ab_j, ab_i)\}$. Note that the first two are due to assumption 1 whereas the last one is due to assumption 3.

On the other hand, $t_i \xrightarrow{fba} t_j^{-1}$ states that if t_i aborts then the compensating task t_j^{-1} must begin its execution. Therefore, it reflects the following logical relationship: $ab_i \Rightarrow ex_j^{-1}$ which results in the following PSS: $\{(ab_i, ab_j), (cm_i, cm_j), (cm_i, ab_j)\}$. Note that the first two are derived from assumptions 2, 3 and 5 whereas the last by applying assumptions 1 and 4. Since $t_i \xrightarrow{bc} t_j$ and $t_i \xrightarrow{fba} t_j^{-1}$ result in the same possible sets, we say $t_i \xrightarrow{bc} t_j$ is equivalent to $t_i \xrightarrow{fba} t_j^{-1}$. Table 1 lists some RD dependencies and their inverses[¶].

The above formalism can be used to enforce *high-to-low* dependencies as follows: For example, if there exists a *high-to-low* dependency $t_i \xrightarrow{bc} t_j$, since the inverse of "bc" is "fba," and "bc" is strong and non-abortive, we replace the above dependency with $t_i \xrightarrow{fba} t_j^{-1}$, meaning that the compensating task t_j^{-1} will begin if t_i aborts. That is, both t_i and t_j can be executed independently, thus t_j need not wait for t_i thereby eliminating potential covert channels. However, if t_i aborts, a compensating task t_j^{-1} will be started. Indeed, this compensating task

[¶]Since the compensating tasks need to be executed by a trusted subject (e.g., a human user), the no-force-commit assumption can be ignored here.

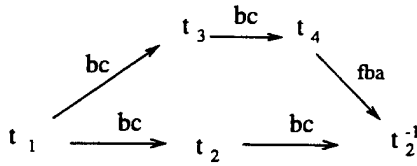


Figure 6: Modified Workflow after executing *compensate_task* for Example 2

must be executed by a trusted subject, e.g., a human user. If there are any dependencies involving t_j , (e.g., $t_j \xrightarrow{x} t_k$), compensating t_j requires t_k to be compensated to capture the cascaded compensation.

Since our approach does not compromise security yet can enforce equivalent compensating dependencies if all tasks are compensatable, thus ensures SSSC-level execution. Figure 6 shows the modified workflow for example 2. A proof and justification for Table 1 can be found in [3].

The following algorithm shows how the above formalism can be employed to modify a *high-to-low* CF-RD type of dependency by introducing a compensating task.

Algorithm 3 [*compensate_task*]

```

for every  $t_i \xrightarrow{x^{CF-RD}} t_j$  in  $W$ 
  if there exists a  $t_j^{-1}$ 
    remove  $t_i \xrightarrow{x} t_j$  from  $W$ 
    if  $x$  is strong non-abortive or weak abortive
      add a node  $t_j^{-1}$  and
      edges  $t_i \xrightarrow{x^{-1}} t_j^{-1}$  and  $t_j \xrightarrow{bc} t_j^{-1}$  in  $W$ 
      /*compensate the low task and add a new inverse
      dependency of the original high-to-low dependency
      from parent to the compensating task of child */
      for every  $t_l$  where  $t_l$  is the closest- $L(t_j)$ -ancestor of
       $t_i$ 
        add an edge  $t_l \xrightarrow{bc} t_j$  in  $W$ 
      end{for}
    for every  $t_j \xrightarrow{y} t_k$ 
       $parent \leftarrow j$  and  $child \leftarrow k$ 
      execute cascaded-compensation
    end{for}
  end{for}

cascaded-compensation
for each  $t_{parent} \xrightarrow{z} t_{child}$ 
  if there exists a  $t_{child}^{-1}$ 
    add a node  $t_{child}^{-1}$  and an edge  $t_{parent} \xrightarrow{z^{-1}} t_{child}^{-1}$  to  $W$ 
     $parent \leftarrow child$ 
  end{for} □

```

Theorem 2 Let W be a workflow. Let W' be the workflow obtained from W by applying algorithm 3 to W . If for every dependency $t_i \xrightarrow{x} t_j$ in W where x is strong

non-abortive or weak abortive, there exists a t_j^{-1} and a t_k^{-1} where t_k is a descendent of t_j , then W' covers W . □

Weak non-abortive type. According to the no-force-commit assumption, the weak non-abortive type dependencies in CF-RD category can only have the following scenarios:

- (1) a “can” relationship on the commit primitive of the child task such as commit dependency (t_j can commit if t_i commits).
- (2) a “must” relationship on the begin primitive of the child task such as weak begin-on-commit dependency (t_j must begin if t_i commits).

In dealing with such a high-to-low dependency, we can simply ignore it without degrading the correctness level because these dependencies result in a *PSS* with all possible combinations of the states of the two tasks. For example, the weak begin-on-commit dependency implies the logic relationship, $cm_i \Rightarrow b_j$, which results in a $PSS = \{(cm_i, cm_j), (cm_i, ab_j), (ab_i, cm_j), (ab_i, ab_j)\}$. In other words, it is the same as if two tasks are without any logic relationship.

5.2.3 Enforcing CN-RD type high-to-low Dependencies

As noted earlier, we treat CN-RD as a combination of CN-RI and CF-RD. Therefore we use both split-task and compensate-task, called *split_compensate_task*. The *split_compensate_task* approach consists of two steps. In the first step, the dependency is treated as if it is a CN-RI type in which the high task is split using *split_task* algorithm. In the second step, the CF-RD dependency between the high partition of the parent task and the child task is handled using the *compensate_task* algorithm. Algorithm 4 formally presents the above illustration.

Algorithm 4 [*split_compensate_task*]

```

for each control-flow dependency  $(t_i \xrightarrow{x} t_j)$  in  $W$  such
that  $L(t_i) \not\leq L(t_j)$ 
  if ( $x$  is of type CN-RD), then
    re-label  $x$  as CN-RI;
    execute algorithm split_task;
     $t_i \leftarrow t_i - t_i^*$ ;
    add an edge  $t_i \xrightarrow{x} t_j$ 
    and label  $x$  as CF-RD;
    execute algorithm compensate_task;
  end{for} □

```

In figure 7, we summarize how each type of dependency can be redesigned.

5.3 Algorithm for the Execution of MLS Workflows

Given a workflow specification W , in the following, we present an algorithm to derive the a *workflow execution*

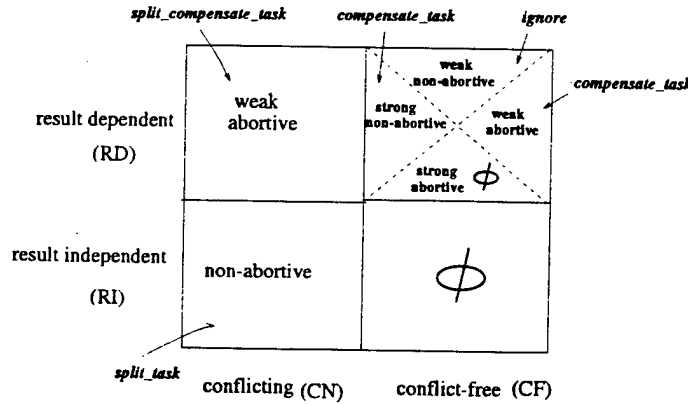


Figure 7: Approach for redesigning each type of dependency

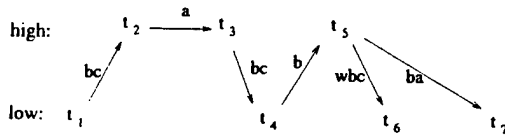


Figure 8: An example workflow (W)

graph (WEG) that determines the execution order of the tasks in a workflow. Here we assume that if there exists a task dependency that is only of one type.

Algorithm 5 [Constructing WEG from W]

nodes of WEG are tasks in W;
include all dependencies $(t_i \xrightarrow{x} t_j)$ in W as edges of WEG such that $L(t_i) \leq L(t_j)$
for each control-flow dependency $(t_i \xrightarrow{x} t_j)$ in W such that $L(t_i) \not\leq L(t_j)$
 if $(x$ is of type CF-RD), then
 if $(x$ is a weak abortive or strong non-abortive dependency), then
 execute algorithm *compensate_task*;
 else ignore x ;
 elseif $(x$ is of type CN-RI), then
 execute algorithm *split_task*;
 elseif $(x$ is of type CN-RD), then
 execute algorithm *split_compensate_task*;
end{for} □

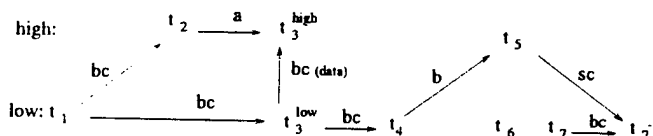


Figure 9: The WEG for W in figure 8

Theorem 3 Let W be a workflow. Let W' be the workflow obtained by applying algorithm 5 to W . Then, W' covers W . □

The workflow execution graph thus constructed consists of all dependencies from low-to-high unless it is to a compensating task. Since all compensating transactions that involve a high-to-low dependency are executed by trusted subjects, enforcing the dependencies in WEG does not cause any covert channels. Figure 9 shows the WEG for the W in figure 8.

Theorem 4 If W consists of all CN-RI type high-to-low dependencies, then the execution of WEG achieves SSSC-level of execution. □

Theorem 5 If there exists a t_j^{-1} for every dependency $t_i \xrightarrow{x_{CF-RD}} t_j$ in W in case where x is strong non-abortive or weak abortive, then the execution of WEG achieves SSSC-level of execution. □

Theorem 6 If W consists of all CN-RD type high-to-low dependencies, then the execution of WEG achieves SSSC-level of execution. □

5.4 Related Work

Research addressing incorporating multilevel security in workflow management systems is fairly new. Recently Atluri and Huang in [2] have proposed a Petri net based approach which can automatically detect and prevent all task dependencies that can potentially cause covert channels. Since their approach eliminates some dependencies, it cannot guarantee correct execution of multilevel secure workflows.

Several researchers have addressed issues concerning execution models for multilevel transactions, which are relevant to our work. Unlike traditional transactions, a multilevel transaction can read as well as write at multiple security levels. Multilevel transaction execution cannot meet both atomicity and secrecy requirements because aborting a portion of the transaction at a lower security level due to the abort of its higher level counterpart creates information flows that violate multilevel security restrictions. Since multilevel transactions can be modeled in our workflow framework, the solutions we propose in this paper are applicable to this area as well. Some of the earlier solutions deal with this problem by relaxing the atomicity requirements. For example Blaustein et al. [5] have proposed several levels of atomicity, and show that based on the structure of the multilevel transaction, only a certain level of atomicity can be achieved. They have proposed two algorithms, called *Low-First* and *High-Ready-Wait*. In the *Low-First*, single level portions (called sections) are executed in the order of increasing security level. That is, all lower level sections must be executed and committed before a higher

level section starts execution. Thus, this algorithm cannot allow high-to-low dependencies. Thus, Low-First can make no guarantees on the level of atomicity. In High-Ready-Wait, all sections of a multilevel transaction are executed (but not committed) in a high to low order and then committed in a low to high order. Thus, High-Ready-Wait cannot enforce low-to-high dependencies. Moreover it works for only hierarchically ordered security structures and also may cause a limited bandwidth covert channel. Thus, Blaustein et al.'s approach works only if either all dependencies are either low-to-high or high-to-low but does not work if there exist both high-to-low and low-to-high (which is referred as cycles in [5]). Our redesigning approach can be employed to eliminate some of the high-to-low dependencies thus increasing the degree of atomicity that can be guaranteed. Note that earlier researchers have also proposed techniques for the elimination of high-to-low dependencies between sections of the multilevel transaction by rewriting the section [5] or maintaining multiple versions of data [7], but their rewriting of each section requires a careful examination of the semantics of the transaction by a human user, and moreover may not be possible in all cases. Whereas our approach redesigns a workflow by simply examining the read and write operations of the tasks and therefore can be fully automated. Our approach is similar to the cache scheme proposed by Smith et al. [15]. Later, Ammann et al. [1] have also proposed a solution based on semantic atomicity which again requires rewriting of multilevel transactions manually. Moreover, Ammann et al.'s approach is based on the assumption that every dependency from a higher to a lower level task can be converted into a lower to a higher level. In this paper, we characterize all the types of dependencies and show that only certain types (called *conflicting* in section 4) can be converted in such a way. Another advantage of modeling a multilevel transaction as a workflow transaction model is that it allows one to distinguish the various types of dependencies that can occur among the sections of a multilevel transaction. This allows one to identify the sections that require to be executed atomically instead of the entire transaction thereby allowing one to specify relaxed atomicity requirements.

6 Conclusions

Correct execution of multilevel secure workflow requires enforcing all the task dependencies. However, ensuring high-to-low dependencies is difficult because of the inherent conflicts between security and correctness. In this paper, we show how a multilevel secure workflow can be executed in a secure and correct manner. Our approach is based on semantic classification of the task dependencies that examines the source of the task dependencies. We propose algorithms to automatically redesign the workflow in such a way that all task dependencies can be executed without compro-

mising security. Note that execution of workflows can be executed by untrusted commercially available workflow management systems although the redesign algorithm must be trusted. For details of our system architecture, the reader may refer to [3]. Our solutions are directly applicable to another relevant area of research — execution of multilevel transactions in multilevel secure databases since the atomicity requirements and other semantic requirements can be modeled as a workflow. By modeling a multilevel transaction as a workflow transaction model allows one to distinguish the various types of dependencies that can occur among the sections of a multilevel transaction. This allows one to identify the sections that require to be executed atomically instead of the entire transaction thereby allowing one to specify a relaxed atomicity requirements. Our redesign process can be used to increase the degree of atomicity one can guarantee for a multilevel transaction. Note that unlike prior research in this area that requires the redesign based on the semantics and therefore requires a careful examination by a human, our approach can be fully automated.

References

- [1] Paul Ammann, Sushil Jajodia, and Indrakshi Ray. Ensuring atomicity of multilevel transactions. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, May 1996.
- [2] Vijayalakshmi Atluri and Wei-Kuang Huang. An Extended Petri Net Model for Supporting Workflows in a Multilevel Secure Environment. In *Proc. of the 10th IFIP WG 11.3 Working Conference on Database Security*, July 1996.
- [3] Vijayalakshmi Atluri, Wei-Kuang Huang, and Elisa Bertino. An Execution Model for Multilevel Secure Workflows. Technical report, Rutgers University, February 1997.
- [4] A. Biliris, S. Dar, N. Gehani, H.V. Jagadish, and K. Ramamritham. ASSET: a system for supporting extended transactions. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 44-54, Minneapolis, MN, May 1994.
- [5] Barbara T. Blaustein, Sushil Jajodia, Catherine D. McCollum, and LouAnna Notargiacomo. A model of atomicity for multilevel transactions. In *Proc. IEEE Symposium on Security and Privacy*, pages 120-134, Oakland, California, May 1993.
- [6] P.K. Chrysanthis. *ACTA, A framework for modeling and reasoning about extended transactions*. PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, 1991.

- [7] Oliver Costich and Sushil Jajodia. Maintaining multilevel transaction atomicity in mls database systems with Kernelized architecture. In Carl Landwehr and Sushil Jajodia, editors, *Database Security, V: Status and Prospects*, pages 173-189. North Holland, 1992.
- [8] Ahmed K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, California, 1992.
- [9] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, pages 119-153, 1995.
- [10] J. A. Goguen and J. Meseguer. Security Policy and Security Models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11-20, 1982.
- [11] Myong H. Kang and Ira S. Moskowitz. A Pump for Rapid, Reliable, Secure Communication. In *Proc. of the 1st ACM conf. on Computer and Communication Security*, Fairfax, VA, November 1993.
- [12] T. F. Keefe, W. T. Tsai, and J. Srivastava. Multilevel Secure Database Concurrency Control. In *Proc. IEEE 6th Int'l. Conf. on Data Engineering*, pages 337-344, Los Angeles, California, February 1990.
- [13] Marek Rusinkiewicz and Amit Sheth. Specification and Execution of Transactional Workflows. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*. Addison-Wesley, 1994.
- [14] Amit Sheth, Marek Rusinkiewicz, and G. Karabatis. Using Polytransactions to Manage Interdependent Data. *Bulletin of IEEE Technical Committee on Data Engineering*, 16(2):37-40, 1993.
- [15] K.P. Smith, B.T. Blaustein, S. Jajodia, and L. Nottargiacomo. Correctness Criteria for Multilevel Secure Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):32 - 45, February 1996.

A Proofs

Theorem 1 Let W be a workflow. Let W' be the workflow obtained from W by applying algorithm 2 to W . Then, W' covers W .

Proof: Algorithm 2 keeps all the $t_i \xrightarrow{x_{CF-RD}} t_j$ dependencies intact. Thus to prove this theorem we need to consider only $t_i \xrightarrow{x_{CN-RI}} t_j$. We prove that W' covers W by proving all the three conditions in definition 7 are true for all CN-RI type dependencies.

Since all the operations removed from t_i while splitting it are placed in some lower level partition t_i^s by algorithm 2 condition 1 of definition 7 is trivially true.

Let us first prove condition 2 of definition 7. Let $t_i \xrightarrow{x_{CN-RI}} t_j$ be a dependency in W . To prove condition 2 of definition 7, we should prove that applying algorithm 2 does not change the order of the operations conflicting with those of t_i that are either in itself or that belong to an ancestor or descendent of t_i . According to algorithm 2, t_i is split into partitions as per definition 10. Because algorithm 2 adds a dependency $t_i^s \xrightarrow{bc} t_i$, from every t_i^s where $s < L(t_i)$, the order of conflicting operations within the original t_i are preserved in W' .

Because of $t_i \xrightarrow{x_{CN-RI}} t_j$, t_i is an ancestor (in fact parent) of t_j . Thus all operations of t_i precede those of t_j in W . According to algorithm 1, t_i must consist of at least one operation $r_i[d]$ such that $L(d) = L(t_j)$ and there must exist a $w_i[d]$ in t_j . Thus, $t_i^{L(t_j)} \neq \emptyset$. Because algorithm 2 adds an edge $t_i^s \xrightarrow{x} t_j$, it implies that $t_i^{L(t_j)}$ is an ancestor of t_j , i.e., all operations of $t_i^{L(t_j)}$ precede those of t_j . Thus W' preserves the order of conflicting operations between t_i and t_j . If there is any child t_k to t_i , other than t_j , then splitting t_i will not change the order of conflicting operations of t_k and t_i because these operations will only be in t_i (since all other partitions consist of read only operations and if there is any other CN type dependency, that should have been expressed as another dependency) and all the other dependencies $t_i \rightarrow t_k$ are not affected while splitting.

Now we prove algorithm 2 preserves the order of conflicting operations of t_i and its ancestors. Suppose t_k is an ancestor at level $s < L(t_i)$. Then t_i conflicts with t_k only if it has a write operation on a data object d ($L(d)$ must be equal to s) and t_i has a read operation involving the same d . That means $t_i^s \neq \emptyset$. A dependency $t_k \xrightarrow{bc} t_i^s$, would preserve the order of the conflicting operations of t_i and t_k . However, if there exists another ancestor of t_i at level s , say t_l , such that t_l is a descendent of t_k , then a dependency $t_l \xrightarrow{bc} t_i^s$ is enough, instead of $t_k \xrightarrow{x} t_i^s$, to preserve the order of the conflicting operations among t_k and t_i as well as t_l and t_i . Applying the same logic, we can conclude that the order of conflicting operations is preserved if we add a dependency $t_l \xrightarrow{bc} t_i^s$, where t_l is the closest- s -ancestor of t_i . Since t_l does not contain a conflicting operation with any other partition $t_i^{s'}$ where $s \neq s' < L(t_i)$, it is not necessary to add a dependency from t_l to such $t_i^{s'}$. Thus we prove condition 2 of definition 7.

All the tasks in W are also in W' . Moreover the additional tasks added by algorithm 2 are simply read-only tasks, thus they do not affect the commit or abort of the original tasks. Thus $PSS(W) = PSS(W')$. This proves condition 2 of definition 7. Therefore W' covers W . \square

Theorem 2 Let W be a workflow. Let W' be the workflow obtained from W by applying algorithm 3 to W . If for every dependency $t_i \xrightarrow{x} t_j$ in W where x is strong non-abortive or weak abortive, there exists a t_j^{-1} and a t_k^{-1} where t_k is a descendent of t_j , then W' covers W .

Proof: Since algorithm 3 does not remove any operations from any task, condition 1 of definition 7 is trivially satisfied. In case of CF-RD type, since t_i and t_j do not have any conflicting operations, condition 2 of definition 7 is always true.

We prove condition 3 of definition 7 as follows: For every $t_i \xrightarrow{x} t_j$, we assume there exist descendants of t_i , t_{k_1}, \dots, t_{k_n} such that $t_j \xrightarrow{x_1} t_{k_1}, \dots, t_{k_{n-1}} \xrightarrow{x_n} t_{k_n}$. We use the following property.

$$PSS(t_i, t_j, t_{k_1}, \dots, t_{k_n}) = PSS(t_i, t_j) \cup PSS(t_j, t_{k_1}) \cup \dots \cup PSS(t_{k_{n-1}}, t_{k_n}).$$

Consider the case where x is either strong non-abortive or weak abortive. $PSS(t_i, t_j, t_j^{-1})$ covers $PSS(t_i, t_j)$. Similarly, $PSS(t_j, t_{k_1}, t_{k_1}^{-1})$ covers $PSS(t_j, t_{k_1})$ and so on. Thus $PSS(t_i, t_j, t_{k_1}, t_{k_1}^{-1}, \dots, t_{k_n}, t_{k_n}^{-1})$ covers $PSS(t_i, t_j, t_{k_1}, \dots, t_{k_n})$. \square

Theorem 3 Let W be a workflow. Let W' be the workflow obtained by applying algorithm 5 to W . Then, W' covers W .

Proof: This trivially follows from theorems 1 and 2. This is because algorithm 2 does not add any new CF-RD type dependency, algorithm 3 and algorithm 4 do not add any new CN-RI type dependency, thus this process will not be cyclic. \square

Theorem 4 If W consists of all CN-RI type high-to-low dependencies, then the execution of WEG achieves SSSC-level of execution.

Proof: Since in WEG, all the CN-RI high-to-low dependencies are converted by algorithm 2 into low-to-high dependencies, all dependencies can be enforced without introducing any covert channels. Thus execution of WEG achieves SSSC-level execution. \square

Theorem 5 If there exists a t_j^{-1} for every dependency $t_i \xrightarrow{x_{CF-RD}} t_j$ in W in case where x is strong non-abortive or weak abortive, then the execution of WEG achieves SSSC-level of execution.

Proof: Since all the CN-RI high-to-low dependencies are converted by algorithm 2 into low-to-high dependencies, all CN-RI dependencies can be enforced without introducing any covert channels.

If x is not weak and non-abortive, according to algo-

rithm 3, every $t_i \xrightarrow{x_{CF-RD}} t_j$ high-to-low dependency is removed and a new high-to-low dependency $t_i \xrightarrow{x^{-1}} t_j^{-1}$ is introduced in WEG. This new dependency can, however, be added only if t_j^{-1} exists. Since this new dependency is enforced only by a trusted subject it does not introduce any covert channels. Thus execution of WEG achieves SSSC-level execution. \square

Theorem 6 If W consists of all CN-RD type high-to-low dependencies, then the execution of WEG achieves SSSC-level of execution.

Proof: Since in WEG, all the CN-RD high-to-low dependencies can be broken down into a CN-RI type and a CF-RD type. By theorem 4 and theorem 5, the execution of WEG that contains all CN-RD type achieves SSSC-level execution. \square

Task-based Authorization Controls (TBAC): Models For Active and Enterprise-oriented Authorization Management

Roshan K. Thomas
Odyssey Research Associates
301 Dates Drive
Ithaca, NY 14850-1326
rtthomas@oracorp.com

Ravi S. Sandhu
Laboratory for Information Security Technology
ISSE Department -- MS 4A4
George Mason University
Fairfax, VA 22030
sandhu@isse.gmu.edu

ABSTRACT

In this paper, we develop a new paradigm for access control and authorization management, called task-based authorization controls (TBAC). TBAC is particularly suited for emerging models of computing. In particular, this includes distributed computing and information processing activities with multiple points of access, control, and decision-making. TBAC articulates security issues at the application and enterprise level. As such, it takes a "task-oriented" perspective rather than the traditional subject-object one. Access mediation now involves authorizations at various points during the completion of tasks in accordance with some application logic. In contrast, the subject-object view of access control typically divorces access mediation from the larger context in which a subject performs an operation on an object. By taking a task-oriented view of access control and authorizations, TBAC lays the foundation for research into a new breed of "active" security models. TBAC has broad applicability ranging from access control for fine-grained activities such as client-server interactions in a distributed system, to coarser units of distributed applications and workflows that cross departmental and organizational boundaries. Furthermore, the ideas in TBAC can form the basis for enterprise-oriented policy modeling and enforcement tools.

1. Introduction

In this paper, we describe a new paradigm for access control and security models, called task-based authorization controls (TBAC) that is particularly suited for emerging models of computing. In particular, this includes distributed computing and information processing activities with multiple points of access, control, and decision making such as that found in workflow and distributed process management systems.

TBAC differs from traditional access controls and security models in many respects. Instead of having a system-centric view of security, TBAC approaches security modeling and enforcement at the application and enterprise level. Secondly TBAC lays the foundation for a new breed of what we call "active" security models. By active security models, we mean models that approach security modeling and enforcement from the perspective of activities or tasks, and as such, provide the abstractions and mechanisms for the active runtime management of security as tasks progress to completion. Such a task-based approach to security represents a radical departure from classical passive security models such as those based on one or more variations of the subject-object view of security and access control. In a subject-object view of security, a subject is given access to objects in a system based on some permissions (rights) the subject possesses. However, such a subject-object view typically divorces access mediation from the larger context (such as the current state of tasks) in which a subject performs an operation on an object.

Our focus in this paper is on active security models for authorization management and access control in computerized information systems. An authorization is an approval act and manifests itself in the paper world as the act of signing a form. Typically, in the paper world, an authorization results in the enabling of

one or more activities and related permissions. The person granting the authorization usually takes responsibility for the actions that are authorized by the authorization. Also, an authorization, as represented by a signature, has a lifetime associated with it during which it is considered valid. Once an authorization becomes invalid, organizations require that the associated permissions no longer be available. As paper-based systems become computerized, the related authorization procedures will have to become automated. Thus the TBAC approach described in this paper was motivated by this anticipated need to automate authorization and related access controls. In particular, the implementation of TBAC ideas will lead to systems that provide tighter just-in-time need-to-do permissions. Also, the TBAC approach leads to access control models that are self-administering to a great extent, thereby reducing the overhead typically associated with fine-grained subject-object security administration.

To motivate further the need for active TBAC security controls, consider a typical workflow-like electronic procedure for preparing and submitting a patent application. For the purpose of our discussion, we can think of a workflow as a partially ordered set of tasks where each task may involve one or more participants and invoked applications, and also issues one or more operations to access and manipulate various objects. The main activities in the patent workflow are shown in Figure 1.

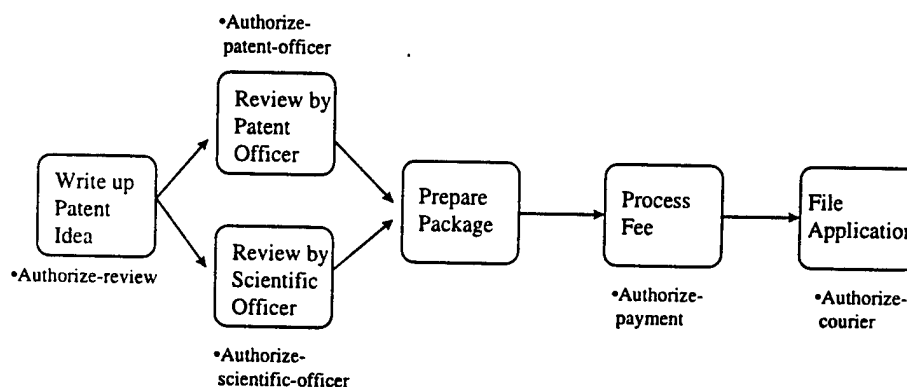


Figure 1. A workflow to process a patent application

The first activity involves someone or a research group writing up the idea for the patent. This will require read and write permissions to a set of electronic documents. Once the writing is done, someone, such as the head of the research group, grants an authorization to review the documents. This authorization does two things. First, it gives a patent officer and a scientific officer the read and write permissions required to review and make corrections. Second, it revokes the write permissions the original authors had so that they cannot make any changes to the documents while the reviews are being done. Once these officers have completed the reviews they grant further authorizations which enable the workflow to progress to the preparation of complete patent application package. At this point, the original authors whose write permissions were revoked, may be granted write permissions again temporarily to make additional corrections. After some deadline, the authors will be prevented from making any more modifications and the workflow then progresses to the fee processing activity. Someone, possibly in the finance or accounts department verifies that there are enough funds in an account to pay for the patent application and then authorizes payment. This authorization grants an accounting clerk a one-time permission to debit the required amount of money from the account and then to issue a check. Finally, the workflow progresses to the last activity which is the filing of the application. Someone authorizes a specific courier to carry the application to the patents office or alternatively authorizes the electronic filing of the application.

In a paper-based implementation of the above workflow, the relevant authorizations manifest as signatures on various forms as they are routed through various departments. At each point, responsible parties are required to manually inspect forms to ensure that the prerequisite signatures (authorizations) are present so that the whole procedure conforms to appropriate organizational policies and checks and balances.

However, when such workflows are automated, we need an active approach to authorization management. The granting, usage tracking, and revoking of permissions needs to be automated and coordinated with the progression of the various tasks. Without active authorization management, permissions will in most cases be "turned on" too early or too late and will probably remain "on" long after the workflow tasks have terminated. This opens up vulnerabilities in systems. Any attempt to minimize such vulnerabilities will require a security administrator to keep track of the progress of the tasks for all enacted workflow instances; an error-prone and impossible task! Thus what is needed is an approach where access control permissions are granted and revoked according to the validity of authorizations and one where this can be done without manual security administration. The authorizations themselves are of course processed strictly according to some application logic and policy. In the remaining sections of this paper we will describe how TBAC ideas can be used to accomplish this.

There are basically two broad objectives guiding our research efforts in TBAC. The first is to model from an enterprise perspective, various authorization policies that are relevant to organizational tasks and workflows. We envision a set of user-friendly tools to help a security officer model and specify policies. Our second objective is to seek ways in which these modeled policies can be automatically enforced at runtime when the corresponding tasks are invoked. We limit the discussion in this paper to the core concepts in TBAC that form our conceptual framework. Various aspects of our research such as languages to model authorization policies, as well as, the runtime mapping of these policies to enforcement mechanisms are topics of ongoing investigation and will be reported in subsequent publications. We also do not address the TCB-style issues related to assurance, as our focus at this point is not on implementation.

Preliminary ideas for TBAC that recognized the need for active security were presented in [3] and [4]. More recently, a workflow authorization model (WAM) was presented in [10]. WAM has the same general motivation as TBAC in that it tries to provide some notions of active security and just-in-time permissions. However, from a conceptual standpoint, TBAC is significantly different and more comprehensive than WAM. In WAM an authorization is a more primitive concept and represents the fact that a subject has a privilege on an object for a certain time interval. In TBAC an authorization (step) has much richer semantics as it models the equivalent of an authorization in the paper world. An authorization act in the paper world may result in the granting of several related permissions. Thus in TBAC an *authorization-step* is a convenient abstraction to model and manage a set of related permissions. TBAC also provides features such as usage tracking of permissions, lifecycle management, the ability to put permissions temporarily on hold without invalidating them, as well as modeling sets of authorizations through composite authorizations.

2. Background: From Passive Subject-object Controls to Active Task-based Security

In this section we discuss how TBAC differs from the traditional subject-object view of access control.

2.1 The subject-object view of access control

Figure 2 illustrates the traditional subject-object view of access control. In the subject-object view, the basic entities are subjects, objects, and the rights possessed by subjects to

Subjects	Objects	
	O1	O2
S1	R	W
S2	W	X

Figure 2. The subject-object view of access control

gain access to the various objects. This can conceptually be represented in an access control matrix such as that in Figure 2. The horizontal and vertical projections of this matrix can be implemented in systems as capabilities or access control lists, respectively. From the standpoint of security models, the subject-object view of access control can be traced to the earlier security models such as the HRU model [8] and its influence can be seen even in later work such as the typed access matrix model (TAM) [9].

A closer examination of the subject-object view of access control will reveal the following characteristics.

- The implicit assumption that there is a central pool of resources to which we need to provide access control.
- Access control information represents isolated units of security information.
- Access mediation is divorced from larger operation context.
- There is no memory of any evolving context associated with past accesses.
- There is no record of the usage of permissions.
- Existing permissions can be revoked but cannot be put on hold.
- Requires fine-grained security administration.

In summary, the subject-object paradigm of access control takes a very system-centric view of protecting a central pool of resources. It enforces a very simple access control discipline which can succinctly be stated as: *if a subject has requested an access operation to an object, and the subject possesses the permission for the operation, then grant the access*. Thus all the access decision function has to check is if the subject has the required permission. However this simplicity is precisely the limitation of subject-object access controls. No other contextual information about ongoing activities or tasks can be taken into account when evaluating an access request (some attempts at access control frameworks to overcome this limitation have been discussed in [14] and [15]). Further, there is no record of the usage of individual permissions. So long as the permission exists, the subject can issue an operation any number of times. We thus consider this to be a "passive" model of security. Next we discuss how TBAC forms an active approach to access control.

2.2 TBAC as an active security model for authorization management

We have coined the concept of active security models to characterize models that recognize the overall context in which security requests arise and take an active part in the management of security as it relates to the progress and emerging context within tasks (activities). Before we elaborate further on TBAC as an active model let us discuss some of the basic ideas in the TBAC approach.

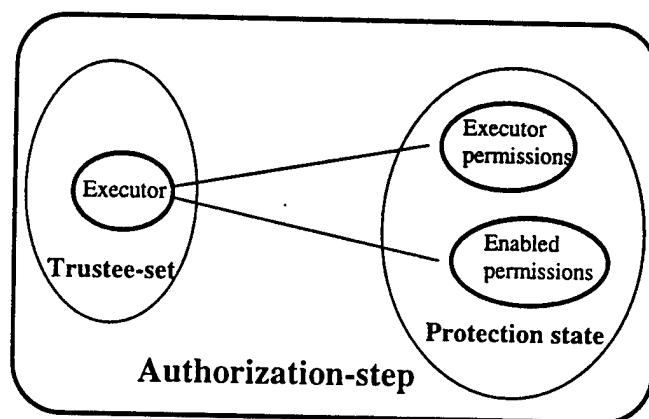


Figure 3. An authorization-step as an abstraction that groups trustees and permissions

One of the most fundamental abstractions in TBAC is that of an *authorization-step*. It represents a primitive authorization processing step and is the analog of a single act of granting a signature in the forms (paper) world. From the standpoint of modeling, it is an abstraction that groups trustees¹ and various sets of permissions, as illustrated in Figure 3. In the paper world, a group of individuals may be potentially allowed to grant a certain type of signature. For example, all sales clerks may be allowed to sign sales orders. However, a single instance of a signature may be granted only by a single individual. For example, sales order 1208 is signed by sales clerk Tom. Similarly, in TBAC we associate an authorization-step with a group of trustees¹ called the *trustee-set*. One member of the trustee-set will eventually grant the authorization-step when the authorization-step is instantiated. We call this trustee the *executor-trustee* of the step. The permissions required by the executor-trustee to invoke and grant the authorization-step make up a set of permissions called *executor-permissions*. Also, in the paper world, a signature also implies that certain permissions are granted (enabled). In a similar fashion, we model the set of permissions that are enabled by every authorization-step. These permissions comprise the *enabled-permissions* set. Collectively, we refer to the union of the executor-permissions and enabled-permissions as the *protection-state* of the authorization-step. Finally, the authority granted by a signature is good only for a limited period of time. Similarly, we associate a period of validity and a lifecycle with every authorization-step.

Classical subject-object

access control

$$P \subseteq S \times O \times A$$

TBAC view of access

control

$$P \subseteq S \times O \times A \times U \times AS$$



 TBAC extensions

Figure 4. Subject-object Versus TBAC views of access control

From the standpoint of access control models, Figure 4 illustrates how the TBAC view of access control differs from classical subject-object access controls. In the latter, a unit of access control or permission information can be seen as an element of the cross product of three domains (sets), namely the set of subjects, *S*, the set of objects, *O*, and the set of actions, *A*. In TBAC, access control involves information about two additional domains, namely, usage and validity counts, *U*, and authorization-steps, *AS*. These additional domains embed task-based contextual information.

In our further discussions, it is useful to be aware of the distinction between an authorization-step class (definition) such as *authorize-review* in the patent workflow definition presented earlier and an authorization-step instance in a particular workflow instance such as *authorize-review* in patent workflow instance with identifier 1234 started at 9AM on Dec 1st, 1996. We use the term authorization-step loosely to mean authorization-step class or authorization-step instance, as determined by context. When the context is ambiguous we will be appropriately precise.

Figure 5 shows the concepts, features, and components that make TBAC an active security model. These include the following:

- the modeling of authorizations in tasks and workflows as well as the monitoring and management of authorization processing and life-cycles as tasks progress;
- the use of both type-based as well as instance and usage-based access control;
- the maintenance of separate protection states for each authorization-step;

¹ We use the term trustee to refer to any one of the following: user, process, agent, service or daemon.

- the dynamic runtime check-in and check-out of permissions from protection states as authorization-steps are processed.

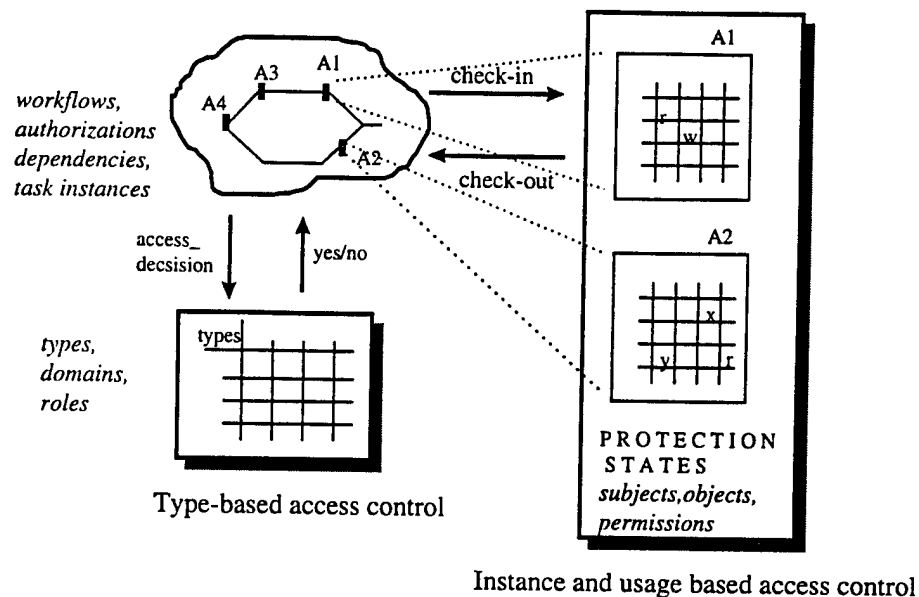


Figure 5. TBAC as an active security model

Every authorization-step maintains its own protection state. The initial value of a protection state is the set of permissions that are turned on (active) as a result of the authorization-step becoming valid. However, the contents of this set will keep changing as an authorization-step is processed and the relevant permissions are consumed. With each permission we associate a certain usage count. When a usage count has reached its limit, the associated permission is deactivated and the corresponding action is no longer allowed. Conceptually, we can think of an active permission as a check-in of the permission to the protection-state and a deactivation of a permission as a check out from the protection state. This constant and automated check-in and checkout of permissions as authorizations are being processed is one of the central features that make TBAC an active model. Further, the protection states of individual authorization-steps are unique and disjoint. What this means is that every permission in a protection state is uniquely mapped to an authorization-step instance and to the task or sub-task instance that is invoking the authorization. This ability to associate contextual information with permissions is absent in typical subject-object style access control models.

The distinction between type-based and instance and usage-based access control is also a significant feature of the TBAC model. Type-based access control is used to encapsulate access control restrictions as reflected by broad policy and applied to types. Instance and usage-based access control on the other hand, is used to model and manage the details of access control and protection states (permissions) of individual authorization instances including keeping track of the usage of permissions.

To elaborate more on how these concepts are used for the active management of authorizations, consider a simple check-voucher processing example that involves the following sequence of authorization steps (for brevity we show only the name of the authorization step and the trustee/role that can request the authorization).

- (1) authorize_prepare_voucher • clerk
- (2) authorize_approve_voucher • supervisor
- (3) authorize_issue_check • clerk

Thus the processing of the voucher involves three phases, namely prepare, approval, and issue. Each phase involves an authorization. As soon as the prepare phase is initiated at the task or workflow layer, there will be an invocation of the first authorization to prepare the voucher (authorization-step 1). This authorization is requested by a clerk, say C. At this point TBAC will utilize type-based access control and an access decision function to check that entities of type "clerk" are allowed to do the "authorize-prepare" operation on vouchers. If this check succeeds TBAC will proceed to check-in (or activate) the required permissions so that the specific clerk, C, (who in this case is the executor trustee) can do the prepare operation. These permissions are checked into the protection state of step 1. As mentioned earlier, we call these permissions *executor-permissions* as they permit the executor to process the authorization. Now, as soon as, clerk C is done with preparing and authorizing the voucher, we consider the `authorize_prepare_voucher` authorization-step to be valid. TBAC will now do two things. First, TBAC will require that previously checked-in executor permissions be checked-out (deactivated) from the protection state of the authorization-step. Next, TBAC may check-in other permissions so as to enable the processing of other activities including the next authorization-step (step 2) which involves authorization for the approval of the voucher by someone in the role of a supervisor. These permissions make up the *enabled-permissions*. During the processing of this second authorization, the supervisor may consume these checked-in enabled-permissions and as a result eventually lead to them being checked-out, and so on. Eventually when step 1 becomes invalid, all enabled-permissions that are still checked-in (active) will be deactivated (checked-out). Finally, when the third authorization-step `authorize_issue_check` is invoked, the organizational policy may dictate a separation of duties requirement. In other words, the clerk that is the executor-trustee of the third step will have to be different from the clerk that was the executor-trustee of the first authorization, `authorize_prepare_voucher`. However, the scope of such a requirement may be limited to only these three authorizations and not to the rest of the authorizations in a workflow. To facilitate such requirements, TBAC supports notions such as *start-conditions* and *scope specifications* that model these kinds of constraints (discussed later).

In summary, TBAC differs from traditional passive subject-object models in many respects by associating the dimension of tasks with access control. First, there is a notion of protection states, which represent active permissions that are maintained for each authorization step. The protection state of each authorization step is unique and disjoint from the protection states of other steps. Each authorization-step corresponds to some activity or task within the broader context of a workflow. Traditional subject-object models have no notion of access control for processes or tasks. Second, TBAC recognizes the notion of a life-cycle and associated processing steps for authorizations. Third, TBAC dynamically manages permissions as authorizations progress to completion. This again differs from subject-object models where the primitive units of access control information contain no context or application logic. Also, TBAC understands the notion of "usage" associated with permissions. Thus an active permission resulting from an authorization does not imply a license for an unlimited number of accesses with that permission. Rather, authorizations have strict usage, validity, and expiration characteristics that may be tracked at runtime. In a typical subject-object access control model, a permission associated with a subject-object pair implies nothing more than the fact that the subject has the permission for the object. There is no recognition or monitoring of the usage of that permission. Finally, TBAC can form the basis of self-administering security models as security administration can be coupled and automated with task activation and termination events.

3. A Family of TBAC Models

Rather than formulating one simple monolithic model of TBAC we have chosen to formulate a family of models. Before discussing the models, we first lay out a framework to guide us in designing the family of models.

3.1 Framework

Our framework consists of formulating a simple model of TBAC called $TBAC_0$ and using this as a basis to build other models.

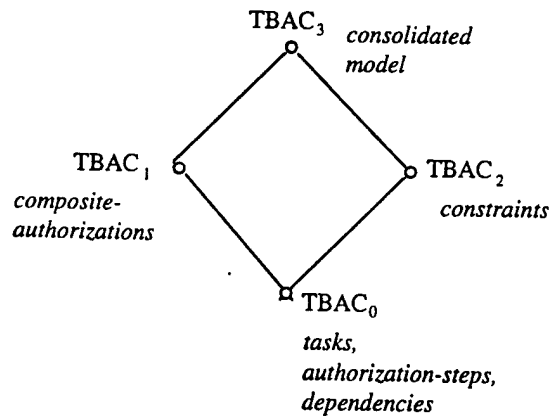


Figure 6. A framework for a hierarchy of TBAC models

Figure 6 shows our framework. $TBAC_0$ is a base model and is thus at the bottom of the lattice. It provides some basic facilities to model tasks, authorization-steps, and dependencies relating various authorization-steps. $TBAC_0$ is a very general and flexible model and is thus the minimum requirement for any system incorporating task-based authorizations. The advanced models $TBAC_1$ and $TBAC_2$ include (inherit) $TBAC_0$ but add more features. $TBAC_1$ incorporates the notion of composite authorizations (discussed shortly) whereas $TBAC_2$ adds constraints. Finally, $TBAC_3$ is the consolidated model that includes $TBAC_1$ and $TBAC_2$ and by transitivity $TBAC_0$.

Formulating such a family of models has many benefits. Researchers and developers can compare their system implementation of TBAC concepts with this family of models. Also, a family of models gives developers various choices in choosing conformance points for their implementations and can thus serve as a guide and evolution path for additional features.

3.2 The model $TBAC_0$

We will now describe the model $TBAC_0$ in more detail. We describe the various attributes or components that make up every authorization-step, followed by its life-cycle, and lastly the dependencies that are used to model authorization policies.

3.2.1 Components of an authorization-step

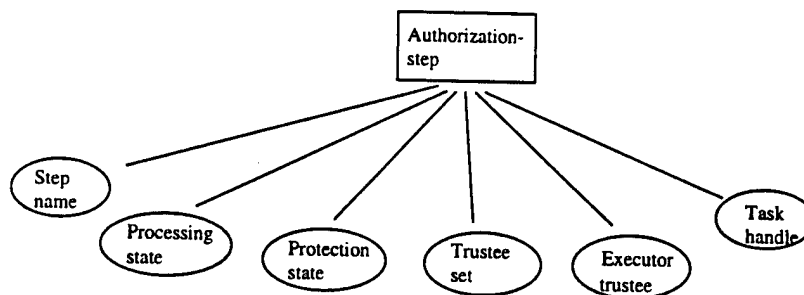


Figure 7. High-level view of the components of an authorization-step

Every authorization-step has to specify a variety of attributes. We now describe briefly each of these attributes (components) in turn.

- *Step-name*: this is the name of the authorization-step.
- *Processing-state*: The current processing state indicates how far the authorization-step has progressed in its life-cycle (discussed shortly).
- *Protection-state*: The protection-state defines all potential active permissions that can be checked-in by the authorization-step. The current value of the protection-state, at any given time, gives a snapshot of the active permissions at the time. Associated with every permission is a validity-and-usage specification. The validity-and-usage-specification specifies the validity and usage aspects of the permissions associated with an authorization-step. It will thus specify how the usage of the permissions will relate to the authorization remaining valid (or becoming invalid).
- *Trustee-set*: This contains relevant information about the set of trustees that can potentially grant/invoke the authorization-step such as their user-identities and roles.
- *Executor-trustee*: This records the member of the trustee-set that eventually grants the authorization-step.
- *Task-handle*: This stores relevant information such as the task and the event identifiers of the task from which the authorization-step is invoked.

Let us now formalize these concepts.

Definition 1. We define a permission, p , as a tuple (s, o, a, u, as) where s stands for the subject or trustee and o represents an object for which the subject is given the right to perform action a u times within an authorization-step instance as . A permission is always associated with an authorization-step instance and its associated protection state (to be explained shortly). If P is the set of permissions, then

$$P \subseteq S \times O \times A \times U \times AS$$

where S is a set of subjects/trustees

O is a set of objects

A is a set of action names

U is the usage and validity specification; a non-zero integer indicating

the number of uses left (the special symbol ∞ is used to indicate unlimited uses) and a flag v indicating if the last usage will make the authorization-step invalid.

AS is the set of authorization-step names.

Definition 2. For each authorization-step instance as , there is an associated protection-state SS_{as} defined by

$$SS: AS \rightarrow 2^P$$

$$SS_{as} = \{(s, o, a, u, as') \in P \mid as' = as\}$$

Definition 3. Each authorization-step instance, as , has a name and the following components:

Processing-state, $PS: AS \rightarrow PS$

Protection-state, $SS: AS \rightarrow 2^P$

Trustee-set, $TS : AS \rightarrow 2^S$
 Executor-trustee $ET : AS \rightarrow S, ET_{as} \in TS_{as}$
 Task-handle $TH : AS \rightarrow T$, where T is a set of tasks

We also state informally two properties.

Property 1. Executor Assignment. For every authorization-step, as , the executor-trustee (ET) component is null until as transitions into the "started" processing state.

Property 2. Non-replaceable Executor. Once an executor trustee is assigned to an authorization-step it is fixed for the entire lifetime of the step.

Property 3. Disjoint Protection States. The protection states associated with various authorization-steps are disjoint. Thus every authorization-step instance has a unique protection state. Thus given a set of authorization-steps a_1, a_2, \dots, a_k , and their respective protection states, p_1, p_2, \dots, p_k , the intersection of two or more of these states will be empty. Formally,

For any p_i and p_j , i is not equal to j , $p_i \cap p_j = \emptyset$

3.2.2 Processing states and life-cycle of authorizations

As mentioned earlier, an authorization is not static; rather it has a lifetime and a life-cycle associated with it. In order to better understand the execution aspects of authorizations, it is useful to consider the various processing states that every instance of an authorization-step goes through during its life-cycle.

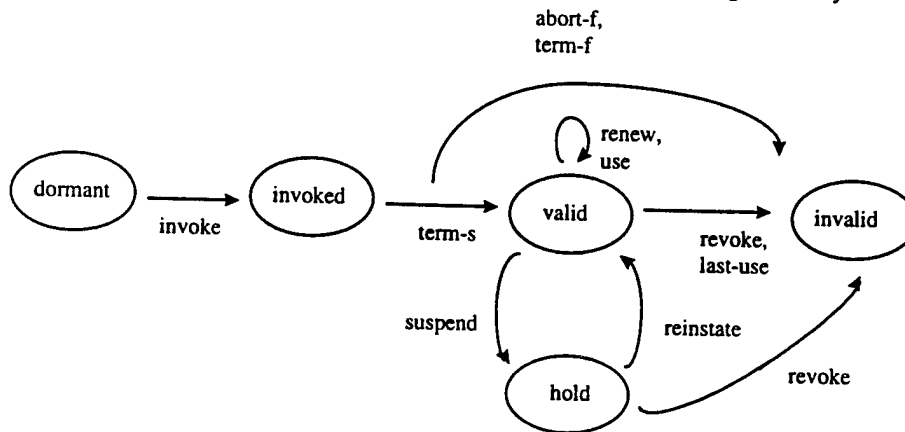


Figure 8. Basic processing states for an authorization-step

A simple view of this life-cycle is to consider every authorization-step instance as going through five states, namely dormant, invoked, valid, invalid, and hold, as shown in Figure 8. An authorization is dormant when it has not been invoked (requested) by any task. Once invoked, an authorization-step comes into existence, and will be processed. If this processing is successful, the authorization-step enters the valid state. Otherwise, it becomes invalid. In the valid state, all associated permissions with the authorization are turned on (activated) and thus available for consumption. From the valid state, an authorization-step will undergo further processing and eventually reach the end of its lifetime and enter the invalid state. Also, a valid authorization-step may be put on hold temporarily. When this happens, all permissions associated with the authorization-step are inactive and cannot be used to gain any access until this hold is released and the

validity reinstated. Eventually, when an authorization becomes invalid, it ceases to exist, and is deleted from the system.

However, to get a more detailed description of what happens to an authorization during its lifetime, one can derive a more elaborate state diagram such as that shown in Figure 9. This more elaborate state diagram recognizes the dimension of usage of permissions. A permission that is in the protection state of an authorization-step is consumed if any action that is enabled by the authorization-step requires the permission. Every operation request thus decrements the usage count of the permission. Once the usage limit is reached an action will no longer succeed as TBAC ensures that the required permission is no longer available.

Figure 9 is a direct refinement of Figure 8. The aborted and started states of Figure 9 are a refinement of the invoked state of Figure 8. Similarly, the valid, hold and invalid states of Figure 8 are each refined into a pair of corresponding used and unused states in Figure 9.

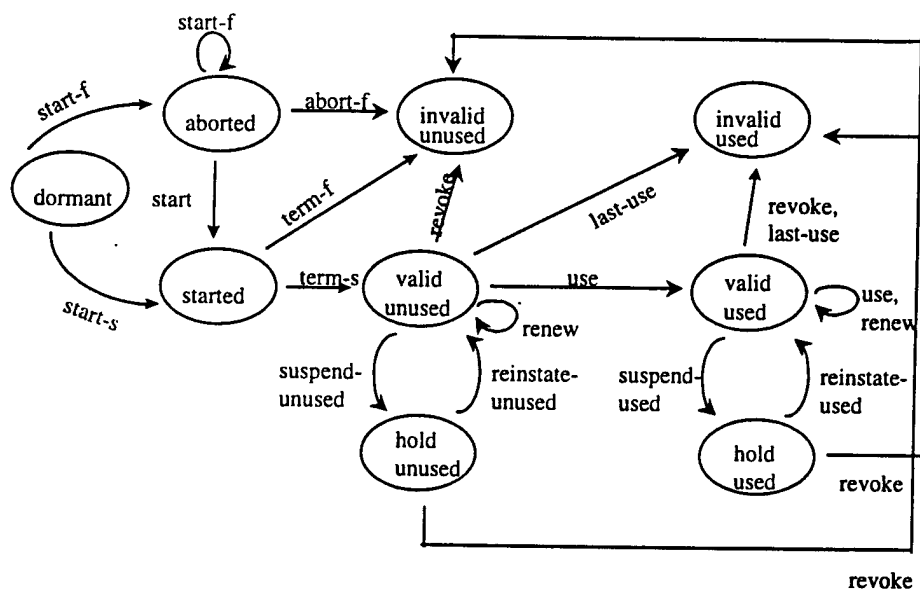


Figure 9. Detailed processing states of an authorization-step

We describe each of the processing states below.

- **Dormant:** An authorization-step is in this state if it has not been invoked by any task. Equivalently, the dormant state can be viewed as one where the authorization-step does not as yet exist. In particular, the protection state of the authorization-step is empty.
- **Started:** Once an authorization-step has been successfully invoked, it enters this state where processing begins.
- **Aborted:** The aborted state is in many ways similar to dormant except that a failed attempt to start the authorization-step was made in this case.
- **Valid-unused:** Once an authorization-step has been started subsequent successful processing will transition it into the valid-unused state.
- **Valid-used:** If an authorization was in a valid-unused state, and it is subsequently used or consumed, then it enters the valid-used state. Depending on policy, an authorization may be used multiple times before it enters the invalid state.

- **Invalid-unused:** This state is entered if certain conditions for an authorization to be valid are not met upon termination or if the authorization had entered the valid-unused state and was subsequently revoked.
- **Invalid-used:** This state is entered either as a result of a last-use transition from the valid-unused state or as a result of a revoke or last-use event (transition) from the valid-used state.
- **Hold-unused:** In this state the unused authorization is temporarily suspended. All associated permissions will thus be inactive.
- **Hold-used:** The authorization is temporarily suspended. All associated permissions will thus be inactive.

We can explain some of the semantics associated with the various states and transitions by considering the sample authorization-step below.

- authorize_prepare_voucher • clerk

In this example, an authorization to prepare a voucher is requested by a user in the role of a clerk. When this step is invoked and an instance of this authorization-step is created, a type-check is made to ensure that the "prepare" permission is allowed between the voucher and clerk type. If this check succeeds, the step transitions into the started state and the executor-permissions are checked-in (activated) into the protection state. Between the started and valid-unused or invalid-unused states, there are no changes in the protection state. Once the step reaches the valid-unused state, the executor-permissions are checked out and the enabled-permissions are checked into the protection state.

These enabled-permissions will allow other actions to continue in the overall workflow. At some point, the authorization-step will become invalid and any remaining permissions in the enabled-permissions set will be checked out (deactivated).

3.2.3 Basic dependencies to construct authorization policies

In the previous sections, we discussed authorization-steps. However, in any application or workflow logic, authorization steps do not stand in isolation. Rather, they are often related and dependent on each other due to policy implications. We now discuss various dependencies and constructs that relate authorization-steps to each other and constrain their execution and behavior. These dependencies can thus be used to formulate enterprise-oriented authorization policies.

We specify dependencies in terms of existential, temporal, and concurrency relationships that hold between events (or states resulting from the occurrence of events). Given an authorization-step A, we use the following notation for the various states of A:

- A^d : the dormant state
- A^s : the started state
- A^a : the aborted state
- A^v : the valid-unused state
- A^{v+} : the valid-used state
- A^i : the invalid-unused state
- A^{i+} : the invalid-used state
- A^h : the hold-unused state
- A^{h+} : the hold-used state

We list the dependency types and their meanings (interpretations) below.

1. $A1^{state1} \rightarrow A2^{state2}$: if A1 transitions into state1, then A2 **must** also transition into state2.

2. $A1^{state1} < A2^{state2}$: if both A1 and A2 transition into states state1 and state2 respectively, then A1's transition must occur **before** A2's
3. $A1^{state1} \# A2^{state2}$: A1 **cannot** be in state 1 concurrently when A2 is in state2.
4. $A1^{state1} \parallel A2^{state2}$: A1 **must** concurrently be in state1 when A2 is in state2.

The first two dependency types \rightarrow and $<$ express existential and temporal predicates and as such are best interpreted as predicates between transition events that lead to changes in the processing states of authorization-steps. They were originally proposed by Klein in [6] to capture the semantics of database transaction protocols. The other dependencies express concurrency properties.

In a later subsection, we will illustrate the use of these dependencies with an order-processing example. Let us now formalize the concept of dependencies.

Definition 4. We define a dependency type as one of the following: \rightarrow , $<$, $\#$, or \parallel and DT the set of dependency types as $\{\rightarrow, <, \#, \parallel\}$;

Definition 5. We define a dependency instance d as a tuple $(a1, dt, a2)$ for which an assignment relation holds from $a1$ to $a2$. If D is the set of dependencies, then

$$D \subseteq AS \times DT \times AS$$

3.2.4 Formal characterization of $TBAC_0$

We now formally define model $TBAC_0$ as follows.

Definition 6. The $TBAC_0$ model consists of the following:

- AS, a set of authorization steps;
- SS, a set of protection states;
- P, a set of permissions;
- D, a set of dependency instances;
- astep: $SS \rightarrow AS$, a function mapping each protection-state to a single authorization-step;
- pstate: $AS \rightarrow SS$, a function mapping each authorization-step to a single protection state.

3.3 The model $TBAC_1$ to support composite authorizations

The model $TBAC_1$ supports the notion of composite authorizations. A *composite authorization* is an abstraction that encapsulates two or more authorization-steps. This is convenient when an authorization-step is too fine-grained a unit to express authorization requirements at a high (abstract) level.

For example, consider the authorization to transfer funds from one bank account to another. Such an action typically requires two authorizations. The first authorization is for withdrawal of funds from the source account and the second to deposit funds into the target account. However, it is useful for modeling purposes

to think of a more composite abstraction called "authorize-transfer" that consists of the individual authorization-steps.

Thus a composite-authorization consists of a set of component authorization-steps. These component authorization-steps can be related to other steps within the *same* composite-authorization through various dependencies. In other words, the authorization-steps of a composite-authorization are not visible externally to other authorization-steps outside the composite-authorization. The motivation for this restriction comes from a desire to follow sound software-engineering principles, especially those related to encapsulation and information hiding. Thus to the external world, a composite-authorization is a single abstraction.

Collectively, the above properties and restrictions impose different semantics during the lifetime of a composite-authorization. In particular, we have to reexamine the notions of when we consider a composite-authorization to be started, valid, and invalid. We approach these issues by associating a *critical-set* of component authorization-steps with every composite-authorization. The critical-set is a subset of the total number of component authorization-steps. We consider a composite-authorization to have started when any member of the critical-set has reached the started state. To be considered valid, all steps in the critical-set have to reach their respective valid states. On the other hand, a composite-authorization is considered invalid as soon as any step in the critical-set becomes invalid.

In addition to the validity associated with the critical-set, a composite-authorization may declare other non-critical-sets of authorization-steps to capture additional states of validity. However, these other sets can become valid only when the critical-set itself is valid and can remain valid only as long as the critical-set remains valid. Collectively, the critical-set along with the various non-critical sets, define progressive states (checkpoints) of validity. The specification of a critical-set within a composite-authorization should thus be done with careful thought given to some minimal notion of validity that ensures consistency with authorization policies for the enterprise.

3.4 The model TBAC₂ and constraints

As mentioned earlier, TBAC₂ supports more advanced notions of constraints. Thus TBAC₂ would be more suitable for an organization that finds TBAC₀ to be too open-ended or not having tight enough controls.

We classify constraints as static or dynamic constraints. Static constraints are those that can be defined and enforced when authorization-steps are specified. Dynamic constraints on the other hand, are those that can be evaluated only at runtime as authorization-steps are being processed.

In TBAC₂ the basic structure of an authorization-step has two components in addition to those present in TBAC₀. We describe these below.

- Start-condition (SC). This component can be used to specify a rich set of constraints that govern whether an authorization-step can transition into the started state.
- Scope (SP). This component controls the visibility of an authorization-step with respect to other authorization-steps when formulating and enforcing authorization policies. Thus scope can be used to control if an authorization-step is visible to an entire workflow, a task, or other finer units such as sub-tasks.

We are currently investigating other static constraints for authorization-steps such as:

- *Constraints on processing state*: This constraint can be used to remove certain processing states (such as hold) from the life cycle of a step.
- *Constraints on protection state*: This can be used to constrain the permissions that are allowed in the protection state (i.e. activated by the step).
- *Constraints on trustee-set*: This can be used to constrain the type as well as the instances of the trustees that can belong to this set. For example, we may want to constrain that the trustee be of type role and limited to instances of project managers and supervisors.

- *Constraints on executor permissions:* This can be used to specify what permissions are not allowed to be among the operating permissions.

The most obvious examples of dynamic constraints are those involving dynamic separation of duties/roles and coincidence of roles. Consider the following four authorizations (for brevity we show only the step name and the trustee-name specified in terms of roles).

A1: auth_prepare_check • clerk
 A2: auth_approve_check • supervisor
 A3: auth_issue_check • clerk
 A4: auth_reapprove_check • supervisor

To prevent fraud and implement various checks and balances, the enterprise policy may dictate that the clerks performing steps A1 and A3 be distinct (separation of duties) while the supervisors for steps A2 and A4 be the same. However, since any clerk or supervisor in the enterprise may be allowed to first perform A1 and A2 respectively, these constraints can be evaluated only at runtime. TBAC₂ allows for the specification of such dynamic constraints. These dynamic constraints are evaluated by looking at the history of the executor trustees in the authorizations that have been invoked. TBAC₂ also allows considerable modeling flexibility by allowing the reach of such dynamic constraints to be influenced by other static constraints such as scope. Thus we may specify that a dynamic separation of duties requirements hold across the scope of a sub-task, task, or other coarser units. By keeping track of the executor trustees of invoked authorizations and combining the notions of dependencies and scope, the TBAC₂ model can be used to provide a much more powerful and general approach to specifying separation of duties requirements than transaction control expressions (proposed in [11]).

4. Conclusions and Summary

We have described an active approach and a family of models for authorization management, collectively called task-based authorization controls (TBAC). Our approach differs from passive subject-object models in many respects. Permissions are controlled and managed in such a way that they are turned-on only in a just-in-time fashion and synchronized with the processing of authorizations in progressing tasks. An authorization-step is a fundamental abstraction in TBAC and is used to group and manage a set of related permissions. To enable this, TBAC supports the notion of a lifecycle for an authorization-step. Further, TBAC keeps track of the usage and consumption of permissions, thereby preventing the abuse of permissions through unnecessarily and malicious operations. TBAC provides for the modeling of enterprise-oriented authorization policies using dependencies that relate authorizations according to some enterprise policy. Our long-term goal is to develop a variety of tools for the modeling and enforcement of authorization policies. The modeling tools will enable a security officer to formulate as well as modify authorization policies using the paradigm of visual languages for interaction. Enforcement will be achieved through authorization servers that load stored policies and enforce the policies at runtime when tasks and workflows are invoked.

We are currently investigating several issues. The consolidated model TBAC₃ needs further examination. In particular, the interaction of composite-authorizations from TBAC₁ and constraints from TBAC₂ requires further study. We are also looking at formulating higher level modeling constructs for authorizations that can be composed from the five types of dependencies mentioned in the paper. For example, it might be useful to have a construct to express atomicity semantics on the validity of a set of authorizations. Also of interest is a framework to cohesively model and understand various constraints. From the standpoint of building end user tools, we are exploring various aspects of visual languages and in particular visual metaphors and related policy grammars to be used by end user tools to express authorization policies. The mapping of policy sentences to dependencies and various security rules that will be automatically incorporated into workflow task definitions is also under investigation. Also under investigation are issues related to the delegation and revocation of authorizations and their related permissions.

Acknowledgment

Some aspects of this research were completed under DARPA contract F30602-95-C-0285. We are grateful to Theresa Lunt and Gary Koob for their support and encouragement.

References

- [1] R. Strens and J. Dobson, How Responsibility Modeling leads to Security Requirements. Proceedings of the Second New Security Paradigms Workshop, Little Compton, Rhode Island, IEEE Press, 1993.
- [2] L.J. LaPadula and J.G. Williams. Towards a Universal Integrity Model. Proceedings of the IEEE Computer Security Foundations Workshop, New Hampshire, IEEE Press, 1991.
- [3] R.K. Thomas and R.S. Sandhu. Towards a Task-based Paradigm for Flexible and Adaptable Access Control in Distributed Applications. Proceedings of the Second New Security Paradigms Workshop, Little Compton, Rhode Island, IEEE Press, 1993.
- [4] R.K. Thomas and R.S. Sandhu. Conceptual Foundations for A Model of Task-based Authorizations. Proceedings of the IEEE Computer Security Foundations Workshop, New Hampshire, IEEE Press, 1994.
- [5] S.K. Chang et. al. Visual-Language System for User Interfaces, IEEE Software, March, 1995.
- [6] J. Klein. Advanced Rule Driven Transaction Management. Proceedings of the IEEE Compcon Conference, 1991.
- [7] D.E. Bell and L.J. LaPadula. Secure Computer Systems: Unified exposition and multics interpretation. EDS-TR-75-306, Mitre Corporation, Bedford, MA, March 1976.
- [8] M.H. Harrison, W.L. Ruzzo and J.D. Ullman. Protection in Operating Systems. Communications of the ACM, 19(8), pages 461-471, 1976.
- [9] R.S. Sandhu. The Typed Access Control Model, Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, CA, May 1992, pages 122-136.
- [10] V. Atluri and W. Huang. An Authorization Model for Workflows, Proceedings of the Fourth European Symposium on Research in Computer Security, Rome, Italy, September pages 25-27, 1996.
- [11] R.S. Sandhu. Transaction Control Expressions for Separation of Duties, Proceedings of the Fourth Computer Security Applications Conference, pages 282-286, 1988.
- [12] M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows, In Modern Database Systems: The Object Model, Interoperability, and beyond, W. Kim, Ed., Addison-Wesley / ACM Press, 1994.
- [13] D. Georgakopoulos, M. Hornick, and A. Sheth, An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure, Distributed and Parallel databases, Vol. 3, pages 119-153, 1995.
- [14] M. Abrams, K. Eggers, L. LaPadula, and I. Olson. A Generalized Framework for Access Control: An Informal Description, Proceedings of the 13th NIST-NCSC National Computer Security framework, 1990, pages 135-143.
- [15] M. Abrams, J. Heaney, O. King, L. LaPadula, M. Lazear and I. Olson. Generalized Framework for Access Control: Toward prototyping the Orgcon Policy, Proceedings of the 14th NIST-NCSC National Computer Security framework, 1991, pages 257-266.

Alter-egos and Roles — Supporting Workflow Security in Cyberspace

Ehud Gudes¹, Reind P. van de Riet²,
Hans F.M. Burg² and Martin S. Olivier³ *

Abstract

Workflow Management (WFM) Systems automate traditional processes where information flows between individuals. WFM systems have two major implications for security. Firstly, since the description of a workflow process explicitly states *when* which function is to be performed by *whom*, security specifications may be automatically derived from such descriptions. Secondly, the derived security specifications have to be enforced. This paper considers these issues for a Cyberspace workflow system by describing a small, but comprehensive example.

The notion of an Alter-ego is central in this description: Alter-egos are objects that represent individuals in Cyberspace (and not merely identify them). In Cyberspace, documents in a workflow system therefore flow between Alter-egos, rather than between individuals.

Keywords

Security and Database systems, Workflow, Cyberspace, Object-Oriented Databases, Role-based security

1 Introduction

Workflow Management (WFM) Systems automate traditional processes where information flows between individuals. Although WFM systems have been in existence for a number of years, the trend towards greater interconnection will greatly impact such systems. On the one hand, interaction will involve more and more nonhuman participants. On the other hand the participants in workflow processes will become more and more unrelated. To illustrate the latter trend, consider the following 'generations' of workflow systems:

*1) Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, Israel, e-mail: ehud@bengus.bgu.ac.il; 2) Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, e-mail: {vdriet, jfmburg}@cs.vu.nl; 3) Department of Computer Science, Rand Afrikaans University, Johannesburg, South Africa, e-mail: molivier@rkw.rau.ac.za

1. All individuals who take part in a workflow process are typically part of the same organization.
2. Individuals who take part in a workflow system are not necessarily members of the organization who 'owns' the WFM system, but are registered with that organization.
3. Participants in a workflow process may never have had any contact prior to participating in the same workflow process.

The key to secure implementation of later generation WFM systems is proper authentication and authorization of participants in a workflow process. It is our contention that Alter-egos (see the next section) are particularly suitable for authentication, while roles are particularly suitable for authorization. Stated differently: we will assume that a potential participant will present an Alter-ego that will serve as proof of the participant's identity.

The intention of this paper is to study the derivation of security rules from a WFM design tool and to consider how such rules may be implemented. Of particular concern is the distinction between the individual, represented by an Alter-ego, and the role in which the individual acts.

The paper is structured as follows: The following section gives some background on Alter-egos, workflow and security. Section 3 introduces the insurance claim example that is used in this paper and shows how security specifications may be derived from the workflow specification. Section 4 discusses an implementation strategy for a workflow process. Section 6 contains the conclusions of the paper.

2 Background

2.1 Alter-egos

Individuals, either in an office environment, or in their homes, will be represented in Cyberspace by objects, called Alter-egos, in the sense of Object-Oriented Technology, and may be considered a combination of Social Security Number and e-mail address. They were introduced in [van de Riet & Gudes96], where it was shown how these Alter-egos can be structured and how Security and Privacy (S&P) aspects can be dealt with. Questions around Responsibility and Obligations of Alter-egos have been discussed in [van de Riet & Burg96a, van de Riet & Burg96b].

The use of Alter-egos to provide high-level security has been discussed in an earlier paper [van de Riet & Gudes96]. The main idea was that if the underlying communication system of Cyberspace ensures that every message contains an unforgeable Alter-ego of the sender (or initiator), one can design more powerful and higher level protection mechanisms than those existing today and which rely mainly on encrypting messages.

2.2 Workflow

In Workflow management (WFM) applications there are tasks to be completed by some organization, but the organization procedures require that this task will be carried out in steps where each step is executed by a different individual and no step can be performed before the steps it depends on are completed [Georgakopoulos95]. We shall demonstrate a certain WFM-tool, COLOR-X, developed by the group in Amsterdam to model Information and Communication Systems, using linguistic knowledge, and we will see how S&P rules can be derived from COLOR-X diagrams.

WFM tools are currently being used to specify how people and information systems are cooperating within one organization. There are at least three reasons why WFM techniques are also useful in Cyberspace. First, organizations tend to become multinational and communication takes place in a global manner. Secondly, more and more commerce is being done electronically. This implies that procedures have to be designed to specify the behaviour of the participants. These procedures may be somewhat different from ordinary WFM designs, where the emphasis is on carrying out certain tasks by the users, while in commerce procedures are based on negotiating, promises, commitments and deliveries of goods and money. However, as we will see, these notions are also present in the WFM tool we will use. Thirdly, people will be participants in all kinds of formalized procedures, such as tax paying or home banking.

2.3 Workflow and Security

This being said, how can we derive security and privacy rules from the Work-flow diagrams (WFDs)? Specifying tasks and actions of people working in an organization naturally also involves the specification of their responsibilities [van de Riet & Burg96a, van de Riet & Burg96b, Olivier96]. This is what WFDs usually do. Responsibility implies access to databases to perform certain actions on data of individuals.

A Workflow Authorization Model is proposed in [Atluri & Huang96b]. Authorization Templates are associated with each workflow task and used to grant rights to subjects only when they require the rights to perform tasks. A Petri net implementation model is also given. Where [Atluri & Huang96a] focusses on synchronising workflow and authorization flow, the current paper focusses on the relationship between individuals (represented by Alter-egos) and the roles they occupy in such workflow processes, while in [Atluri & Huang96a] the authors emphasize information-flow issues using a multi-level security model.

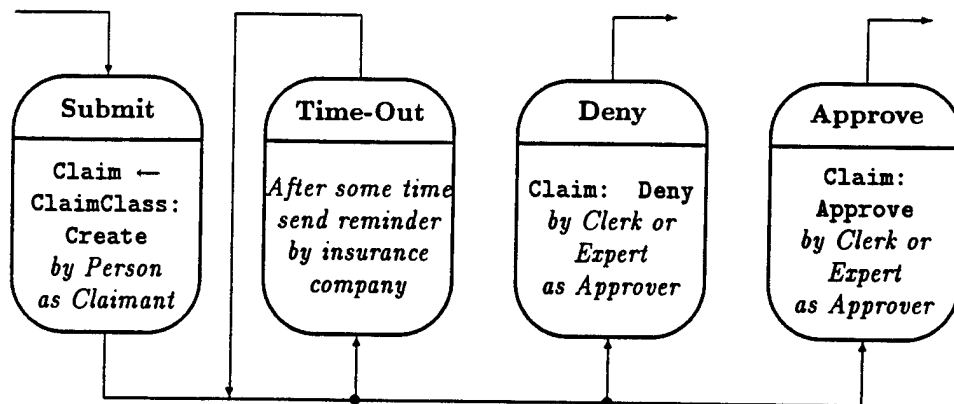


Figure 1: Fragment of a workflow process

2.4 Alter-egos, roles and workflow security

As noted earlier, it is our contention that Alter-egos are particularly suitable for authentication, while roles are particularly suitable for authorization. Not only are Alter-egos suitable for authentication — the nature of the mechanism to represent participants *has* to progress towards full-function Alter-egos. Similarly, the role concept needs to evolve from that required by the earlier generations to that required by the later generations.

Alter-egos required by first generation WFM systems primarily serve to authenticate the user to the system. A user identifier with a password or PIN (personal identification number) may be adequate. In second generation systems the requirements, in addition, include privacy, integrity and non-repudiation. These issues become important since many of the participants are not employed by the organization and are accessing the system from remote systems. In third generation systems these issues are still important, but it also becomes important to construct the Alter-egos such that they can be trusted by the represented individual to act as agent for the individual.

3 The Insurance-claim Application

In this paper the following simple fragment of a workflow application will be used to illustrate the concepts involved: A *claimant* submits an insurance claim, which is either approved or rejected by some *approver*. If the claim is not approved (or rejected) within some specified period, the approver is reminded to give attention to this claim. This fragment is depicted graphically in figure 1. For another example, see [van de Riet & Burg97].

The following requirements are obvious, even from this simple example:

1. The approver has to be duly authorized to approve the claim. The requirements for such authorization depend on the specific application; the mechanisms to enforce such requirements are the concern of this paper.
2. Access to the documents involved in the workflow depend on the roles individuals play in the workflow process.

A specification for the same process, but using a COLOR-X diagram, is given in figure 2. Note that in figure 2 we used the following conventions: A box in figure 2 (a) denotes an entity or type. A line is a relationship and a line with open arrow is an is_a relationship. The circle with an X means exclusion. In figure 2 (b) we have:

- each box of actions has a mode: PERMIT, NEC or MUST. The latter one means an obligation based on some negotiating in the past: as we are not sure that the action is actually carried out within the prescribed time it is necessary to define a counter measure. The mode NEC means we can be sure the action is necessarily carried out by the system. PERMIT is self evident.
- the actions are described in a formal language involving the participants and their roles;
- the lightning arrow denotes a situation in which the conditions in the identification part (denoted by id) are not satisfied. In the diagram we let the reminders go indefinitely, which was done for shortness sake;
- The "approve" and "deny" boxes from figure 1 correspond to the arrows R2="approve" and R2="deny" going out the lowest but one box in figure 2 (b).

We now derive the authorization tuples from the diagrams above. We use the following heuristic rules:

1. If an action involves data in a database, the agent of this action should be authorized to perform the corresponding actions on the database.
2. An action, with modality MUST, involves an obligation to perform a specific action within a prescribed amount of time. This implies that in some database, oblDB, the administration about this obligation is kept. The object which creates the MUST action, i.e. the agent of the action leading to this MUST action, can move the deadline (only shift it to the future of course); that is explicitly not allowed to the object who has to carry out the action. Of course this object can refuse to carry it out, but then penalties may be the result.

3. In a send or reminder action the sender is assumed to write in the message database, while the receiver is assumed to be able to read from it.

The databases involved are:

- For the claims: claimDB;
- For obligations: oblDB; and
- For the messages: messDB.

The syntax we will use to indicate that an actor in some role is authorized to perform a specific operation on a specific database, is as follows:

AUTH <name database, role actor, operation>

From the diagram we thus have the following authorization tuples:

```
AUTH<claimDB, claimant, add>
AUTH<claimDB, insurance_company, read>
AUTH<claimDB, approver, read>
AUTH<claimDB, cashier, create>
AUTH<oblDB, claimant, shift>
AUTH<oblDB, approver, not shift>
AUTH<messDB, insurance_company, write>
AUTH<messDB, approver, read>
AUTH<messDB, cashier, read>
```

Furthermore, from figure 2 (b) we derive that there is a partial ordering with respect to authorization for actions concerning the claimDB, for the following roles:

```
clerk >> approver
expert >> approver
```

According to figure 2 (a) all three roles or types are subtypes of employee. Note also that an approver cannot be the same person as the claimant.

The authorization tuples as derived above may in general conflict, as there are positive and negative tuples. In the case of an approver not being allowed to shift an obligation, we also may have the situation that the approver sets a deadline for someone else, in which case there will appear an authorization tuple:

AUTH<oblDB, approver, shift>

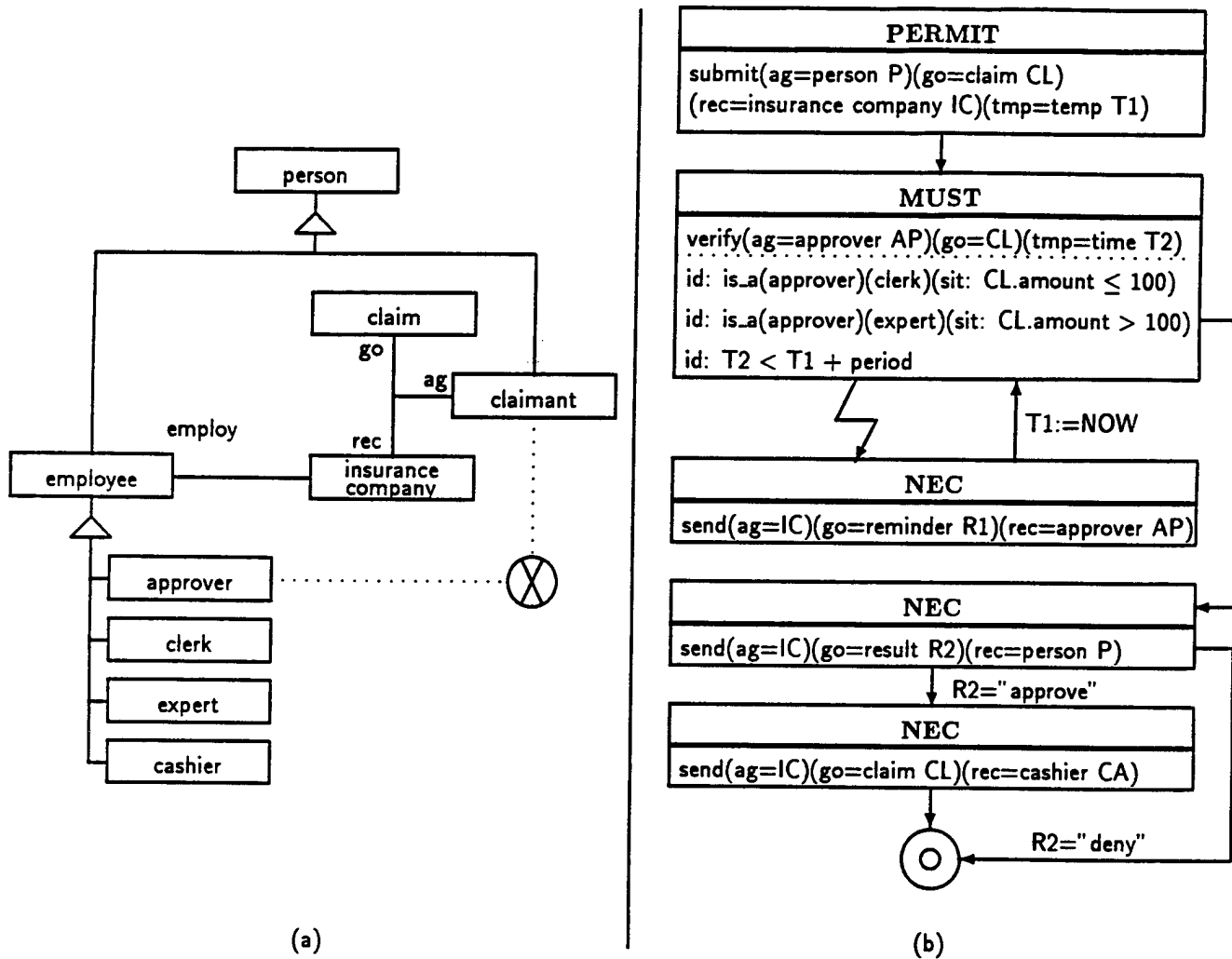


Figure 2: Application specified using a COLOR-X diagram

which is evidently in conflict with the one we mentioned. The reason for this problem is, however, very simple: oversimplification. In actual practice we would also register the specific task to be performed, and not merely allow access to an entire database.

The above security checks represent simple access-control rules which can be automatically derived from the Workflow specifications. However, in reality the situation is much more complex. First, there may be more complex constraints, such as: a claimant and an approver may not be the same individual (alter-ego). The issue of constraints is discussed further in section 4.2. Second, the distinction between a *Role* and *Individual* (Alter-ego) may be important. In some instances, only the role is required to approve access. In other instances, only a specific Alter-ego may approve a claim. This is further discussed in section 4.3. Thirdly, the above rules assume a static state where the agents represented by Alter-egos and Roles are “alive” and respond in time. However, time is an important factor! Rights of some individuals may be time dependent (see also [Bertino94]). In particular, users may change their role in the organization, new roles may need to be created for a particular alter-ego (e.g. the expert role), or deleted. The administration of roles and its dynamics are therefore an important issue. This is discussed in section 4.4.

4 Implementation

The implementation of the above model largely depends on the underlying object system. Below we sketch our approach to implement it in Mokum. Note the fact that the various agents are responsible for performing the required security checks. This avoids the bottlenecks associated with a centralised checking facility in a distributed system. It also provides more flexibility in the types of checks that may be performed over those from a standardised facility. The fact that the checks are interspersed through the code is not a concern because the intention is to automatically derive the code and the checks from the workflow specification. The following discussion concentrate on implementing the run-time access-control, the administration issues are discussed in Section 4.3.

4.1 Implementation based on Mokum

The main idea here is to use the “Collection Keepers” [van de Riet & Gudes96] to execute the customized (knowledge-based) security checks. Some of the Workflow security related pseudo-code performed by each participant (similar to Mokum’s code described in [van de Riet & Gudes96]) is shown below.

Authorization tuples as described earlier, are defined as follows:

```
type authorization_tuple
  has_a alter_ego: thing
```

```

has_a role: thing
has_a database: thing
has_a access_mode: thing

```

The Application Administrator is responsible for performing additional authentication checks and for authorizing operations in the workflow system. In order to do this, the Application Administrator maintains the authorization database and processes the authentication and authorization events. For brevity the details of the code have been omitted.

The insurance company receives the claim and directs it to an approver, which may be a clerk or an expert.

```

type insurance_company is_a thing
script
  at_trigger submission:
    message:role=claimant,
    A = message:claim:amount,
    (A < 100, choose_clerk(AP); choose_expert(AP)),
    add_type(AP, approver),

```

The claimant is the submitter of the claim.

```

type claimant is_a person
script
  at_trigger send_claim:
    create(M, messageT, [(data_base=claimDB), (access_mode=add)]),
    send(application_administrator, authenticate_user,M),
    compose (Claim),
    /* using the incident and the amount of money */
    /* involved by interaction with the user */
    Claim to M:claim,
    send(application_administrator,authorize_user,M),
    /* Note that the authorization goes before the following */
    /* action, so if it fails the next action will not be */
    /* executed */
    send (insurance_company,submission, M).
end_script

```

The approver is the clerk or expert responsible for verifying and approving claims.

```

type approver is_a employee

```

```

script
  at_trigger verify_claim, reminder_message:
    /* note that for the sake of brevity we let the approver */
    /* do the same when it receives a new case or when he    */
    /* receives reminder for some case                        */
    CL=message:claim, C=message:client,
    create(M1, messageT, [(data_base=claimDB),
      (access_mode=read),(claim,message:claim)]),
    send(application_administrator,authorize_user,M1),
    show_claim_on_screen(CL),
    /* now the approver may take a few days to verify the claim*/
    verify_claim(CL, Result),
    create(M2, messageT, [(client=C), (result=Result)]),
    send(insurance_company, receive_from_approver,M2)
end_script

```

4.2 Constraint checking

Although not shown in the example given in figure 2, it is possible to specify constraints in the WFM diagrams. Such constraints may have the form "the approver of a claim must be different from the claimant". These constraints can also be specified in the Mokum scripts of the Alter-egos involved.

Constraints may be specified as pre-conditions, post-conditions or through-conditions. Ideally, constraints are checked by the various choose methods. For example, when the choose_expert method is selected above one could have as a Pre-condition: not AP = sender.

In contrast, if the constraint specifies that some particular approver may not approve claims involving a claimant who has not paid any outstanding fees, this may not be verifiable when the approver is selected, since the approval process may take a significant amount of time, in which the claimant's account may become overdue. Such a condition therefore needs to be specified as a post-condition, to be checked at completion of the approval activity. This could be implemented by adding somewhere, after getting the response of the approver and before sending of messages to cashier and claimant, the following to the code of the trigger submission: `check_due_fees(sender)`.

If the constraint specifies that the approver may not be married to the claimant, the intention may be that they should not be married at any point during the approval process (which may take a while!). It is clear that neither a pre-condition, nor a post-condition (or a combination of the two) fully checks this constraint. (A post-condition is, in principle usable, if a history of marriage(s) is kept; it may, however, postpone handling of the

situation longer than necessary.) We call these constraints *through-constraints*. In our case they could be implemented in the Mokum script of approver, instead of `verify_claim`, we could have: `verify_claim_and_check_spouse`. Of course in the body of this procedure the actual checking needs to be programmed — for example to inspect a database.

4.3 Roles versus individuals

We assume that messages in a workflow system are routed to any qualifying individual operating in an appropriate role. It is, however, possible that in exceptional cases, the message needs to be directed at a particular individual acting in a role. A particular approver may, for example, be requested to deal with a particular claim because of expert knowledge the approver has. Another example is the “not-married” constraint above.

Where previous constraints dealt with avoiding of conflicts, it is also sometimes necessary that people co-operate to perform a given action. Consider a bank safe that is to be opened only after two distinct, authorised subjects have requested it. It is clear that this requirement is easily handled in the current approach by simply requiring the related actions from two ‘keybearers’ (who have to be distinct individuals) within an acceptably short time period (enforced with deadlines). The Alter-ego concept gives us this essential capability of distinction between and identifying certain individuals

4.4 Role administration

As mentioned in section 3, a Workflow system is a very dynamic system. Roles need to be created, deleted changed, etc. The administration of roles is therefore a critical component in the implementation. For that purpose we assume the existence of the Workflow security administrator (called above: Application administrator). This administrator handles the following tasks:

1. It authenticates each new user and creates for her the appropriate role.
2. It is consulted whenever the dynamics of the situation requires the change of roles. For example, if for a particular claim, the Expert role is required, the administrator is consulted to check whether the current Approver’s alter-ego can amplify its role to become an Expert. If not, it checks whether there is already an active Expert role in the system and if there is, sends to the Approver node its identity, and if it does not exist, it will wait for the right alter-ego to be instantiated and authenticated with this Role.

We see the Roles/Alter-ego administration as an essential part of the implementation. This implementation can use the Distributed Mokum architecture described next.

4.5 Distributed Mokum

A first version of Distributed Mokum [Radu, Dehne & van de Riet] is currently being tested. This version of Mokum can be run at several sites. The objects can communicate with 'local' objects and with 'global' objects. An interface has been written in Java which, using the socket mechanism, cares for the necessary communication. Each Mokum program has its own Administrator, which not only takes care of the external communication problems, but also,beit in a very limited fashion, of security problems. It is a kind of Security Administrator. We are still looking at the problem to concentrate such security administrative duties in an applet. In this implementation an applet is being used for the man-machine interface between a user and the Mokum program.

How to implement a Mokum system which is truly distributed, where objects may consist of subobjects residing at different sites and where addressing these subobjects is completely transparent, is still a subject of study.

5 Supporting technologies

In order to securely implement the system described above, it is necessary to ensure that the underlying infrastructure. Secure communication protocols are obviously essential. Before we review such protocols, we briefly consider CORBA to support the implementation described above.

5.1 Implementation based on CORBA

CORBA which is becoming one of the major standards for Object-oriented software systems has recently published the Security reference model specifications [OMG96]. Many of our concepts map directly into the CORBA architecture. Our Alter-ego will represent a "principal" in CORBA. In terms of protecting messages and assuring message integrity, CORBA provides some facilities, but in our opinion these facilities are redundant in Cyberspace because of the existence of secure communication protocols (see section 5.2).

The active customized code for each participant will be coded within the "access decision functions" of CORBA, and is enforced automatically by the ORB (Object Request Broker) before any Object invocation. Auditing can also be specified by the CORBA's model and also implemented by the ORB. In addition Non-Repudiation services are also provided for reasons of Accountability. Although Roles as used in this paper are not supported in the CORBA model, other means such as Domains may be used to implement them. The problem of *administration* of authorization information is discussed in the [OMG96], but a full specification is not given.

Since the security reference model is not yet implemented and not even accepted, one can resort to existing CORBA services to implement our model. One such approach can follow the scheme suggested by Sheth in his CORBA based Workflow architecture [Miller et al96]. In this architecture, there is a Task manager associated with every task (object in our case), which executes the interface code defined in the IDL file. We will need to add to such a task the active security checks and the ability to **re-initialize** its parameters each time it receives a message from the Application administrator requiring it.

5.2 Using Secure communication protocols

The recent literature and the World Wide Web has much information about secure communication protocols. Lipp and Hassler [Lipp & Hassler96] present a survey of such protocols. They can be at the message level such as Netscape SSL or Microsoft PCT [PCT96] or depend on the requirements of the application which is HTTP or higher, such as SHTTP. SHTTP supports end-to-end secured transactions which can be initiated symmetrically, so that servers and clients are treated equally with respect to their preferences. Message protection may be provided by signing, authenticating, or encrypting the message, or by applying any combination of these. There are basically two types of messages in the above described Workflow system. There are the standard messages between the participants' Alter-egos. These messages are assumed to contain the Sender's pair (*Role*, *Alter-ego*) and the receiver's Role. The integrity of messages provided by the lower level protocols is sufficient. Once the Alter-ego is authenticated, and a Role is assigned, the pair (*Role*, *Alter-ego*) is assumed to be part of every message (e.g. in its header) and this is sufficient for applying the higher-level customized security checks.

The existence of the pair (*Role*, *Alter-ego*) — actually the triple (Site, Role, Alter-ego) — can also be used by security firewalls. The firewall will be associated with the workflow application and will reject any message which does not have in the header the (*Alter-ego*, *Role*) pair.

Another problem is the protection of documents which may be required by the Expert system to approve the claim. One need to authenticate the validity of these documents and just message protection is not sufficient. A scheme similar to the one suggested to protect documents on the Web, i.e. CCI-PGP scheme can be used for that [Weeks96].

6 Conclusions

This paper considered security in a workflow system. It has been argued that the notion of Alter-egos is central in such workflow systems. The participation of an Alter-ego in each

message enables the complete authentication and some specific individual-based checks that are required in such an environment.

It is clear from the previous section that different implementation alternatives do not only hold different advantages, but are based on different trusted components: Note in particular the trust placed in the various actors in the case of the Mokum implementation.

It is clear that the role concept in workflow system holds particular benefits. Additional research needs to be done to investigate this potential — in particular for third generation workflow systems.

References

- [Atluri & Huang96a] Atluri, V., and Huang, W.K. "An extended petri net model for supporting workflow in a multilevel secure environment," Proc. Annual IFIP WG 11.3 Conf. on Database Security, Como, Italy, August, 1996, pp. 199-216.
- [Atluri & Huang96b] Atluri, V., and Huang, W.K., "An Authorization Model for Workflows," in Bertino, E., Kurth, H., Martella, G., and Montolivo, E. (eds), *Computer Security — ESORICS 1996*, Springer, 1996, pp. 44-64.
- [Bertino94] Bertino, E., Bettini, C., and Samarati, P., "A Time-based Authorization Model," Proc. ACM Int. Conf. on Computer and Communication Security, Fairfax, Va, Nov. 1994, pp. 126-135.
- [Georgakopoulos95] Georgakopoulos, D., Hornick, M., and Sheth, A., "An overview of workflow management: from process modelling to workflow automation infrastructure," *Distributed and Parallel Databases*, Vol 3, No. 2, 1995, pp. 119-154.
- [Lipp & Hassler96] Lipp, P., and Hassler, V., "Security concepts for the WWW," Proc. 2nd Int. Conf. on Communication and Multi-media security, Essen, Germany, 1996, pp. 85-95.
- [Miller et al96] Miller, J.A., Sheth, A.P., Kochut, K.J., and Wang, X., "CORBA-based run-time architecture for Workflow management systems," *Journal of Database Management*, Vol 7, No. 1, Winter, 1996, pp. 16-27.
- [Olivier96] Olivier, M.S., "Using workflow to enhance security in federated databases," Proc. 2nd Int. Conf. on Communication and Multimedia Security, Essen, Germany, 1996, pp. 61-72.

- [OMG93] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG Document number 93.12.1, December, 1991.
- [OMG96] Object Management Group, URL "<http://www.omg.org:80/docs/orbos/>". Documents: 96-08-03.ps, 96-08-04.ps, 96-08-05.ps, and 96-08-06.ps.
- [Radu, Dehne & van de Riet] Radu, S., Dehne, F., and Van de Riet, R.P., *A first step towards distributed Mokum*, Technical Report 428, Computer Science Department, Vrije Universiteit, Amsterdam, In preparation.
- [van de Riet & Burg96a] Van de Riet, R.P., and Burg, J.F.M., "Modelling Alter-egos in Cyberspace: who is responsible?", Proc. of WebNet 96, San Francisco, 1996, AACE, Charlottesville, USA, pp. 462-467.
- [van de Riet & Burg96b] Van de Riet, R.P., and Burg, J.F.M., "Linguistic Tools for Modelling Alter Egos in Cyberspace: Who is Responsible?" Journal of Universal Computer Science, Vol 2, number 9, Springer, 1996, pp. 623-636.
- [van de Riet & Burg97] Van de Riet, R.P., and Burg, J.F.M., "Modelling Alter-egos in Cyberspace using a Work Flow Management Tool: who takes care of Security and Privacy?", Submitted.
- [van de Riet & Gudes96] Van de Riet, R.P., and Gudes, E., "An object-oriented database architecture for providing high-level security in Cyberspace," Proc. 10th Annual IFIP WG 11.3 Conf. on Database Security, Como, Italy, August, 1996, pp. 92-115.
- [Weeks96] Weeks, J.A., Cain, A., and Sanderson, B., "CCI-Based Web security: a design using PGP," URL "http://sdg.ncsa.uiuc.edu/~jweeks/www4/paper/current_rev.html"
- [PCT96] MicroSoft Corp., URL: http://microsoft.com/intdev/security/misf13_4.htm

Architectures and Systems
Chair: Sujeet Sheno

A Two-tier Coarse Indexing Scheme for MLS Database Systems

Sushil Jajodia¹, Ravi Mukkamala² and Indrajit Ray¹

¹Center for Secure Information Systems and
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030-4444, U.S.A.
{jajodia,iray}@isse.gmu.edu

²Department of Computer Science, Old Dominion University
Norfolk, VA 23529-0162, mukka@cs.odu.edu

Abstract

In this paper, we propose a two-tier indexing scheme for multilevel secure database systems, primarily with the intent of improving query response time and reducing the storage required for indexing. At the bottom tier, our scheme requires separate single-level indices, one for each partition of the multilevel relation; at the top tier, our scheme requires a coarse multilevel index consisting of only those key values from the single-level indices that are necessary to direct a query to the appropriate single-level index. Our scheme seems suitable for both single-level and range queries. We prove this claim for B^+ trees by providing a detailed performance analysis for this index structure. We also give the algorithms for inserting and deleting key values, as well as for searching for a key value, in the proposed index structure.

1 Introduction

Database systems often index a relation to make access to the relation's tuples faster than is possible by a sequential scan of the entire relation. An index is created on some *indexing field* (typically the primary key) of the relation; the index file stores each value of the indexing field together with a pointer to the block in the physical storage that contains the record with that field value. The index file being much smaller than the relation, can be searched much faster than the latter.

This efficiency is not always achieved in trusted database management systems (DBMSs) (e.g., Informix OnLine/Secure [4], Trusted Oracle [9], and Sybase Secure SQL Server [10]). This is because they provide only two indexing options: either multiple *single-level* indices, a separate index for data at each security level or a *trusted* multilevel *global* index over all data in the multilevel relation. While the single-level index structure works well for those

```
SELECT  EMP.NAME, EMP.SALARY
FROM    EMP
WHERE   ROWLABEL = 'SECRET'
```

Figure 1. A Single-level Query

queries where users specify the security level of the data (we refer to these as *single-level* queries), it performs poorly if the security levels of the data are left unspecified by the users. A common kind of query is the *range query* in which the user gives a desired range of values for the indexing field and wishes to retrieve all those tuples whose values in the indexing field fall within the desired range; obviously, single-level indices perform dismally in the case of range queries [8]. The problem with the multilevel index is that while it is more efficient for answering range queries than the single-level index structure, it is less efficient for single-level queries. Example single-level and range queries are given in figures 1 and 2, respectively.

In this paper, we propose a two-tier indexing scheme which retains the advantages of both kinds of index structures. We maintain multiple single level indices, one for each security level and construct a *trusted coarse* multilevel index over these single level indices. The coarse index consists of only those key values from the single-level indices that are necessary to direct a query to the appropriate single-level index. Our scheme seems suitable for both single-level queries as well as range queries. To prove this claim, we choose B^+ tree as the indexing scheme, and provide a detailed performance analysis for this index structure.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 contains an example to illustrate the different indexing schemes and their relative merits. Section 4 gives an informal description of the different steps involved in creating and managing our index

```

SELECT  EMP.NAME, EMP.SALARY
FROM    EMP
WHERE   SALARY >= 20000 and
        SALARY <= 50000

```

Figure 2. A Range Query

scheme. (The detailed algorithms have been left out for lack of space.) In sub-section 4.4, we describe a variant of our indexing scheme which is cheaper to use in terms of storage, but sometimes is more expensive in terms of querying. In section 5, we give details of our performance model, and summarize the results of our analysis in section 6. Section 7 concludes the paper with a discussion of our future work.

2 Related Work

For obvious reasons, indexing schemes have received wide attention from database researchers (e.g., see [3, 5, 6]). The notion of B-tree, a variant of B⁺ trees, was first introduced by [1]. Algorithms for searching and insertion and deletion in B-trees and B⁺ trees can be found in [5] while the issue of concurrent operations on B-trees has been dealt with in [2] and [7].

To the best of our knowledge, the only work that deals with indexing for MLS database is by [8]. They maintain multiple single-level indices, one index for each security class. To facilitate range queries, they add *cross links* in the index structure; a cross link is a pointer from a block B_i to another block B_j at a lower security level, signifying that the range of data in B_j contains a subset of the range of data in B_i. Cross links help in the evaluation of a range query by allowing immediate access to the block containing the relevant data in the next index once an initial index has been searched for values in the range.

Although the concept of cross links is novel and seems to speed up the evaluation of range queries, the cost of maintaining the cross links appears to be very high. The authors assume that the security levels in the system form a total order; it seems that for any arbitrary partial order of security classes the overhead of maintaining cross links may be prohibitive. In fact, the authors acknowledge that it is impossible to predict whether indexing with cross links performs better or worse than the single-level index structures.

3 Motivation for Our Approach

Consider the multilevel index file *F* shown below:

Key Values: 5(TS), 7(TS), 8(TS), 9.5(TS), 10(S),
12(S), 15(C), 13(C), 7.5(C), 7.3(C),
20(S), 21(S), 22(S), 9(TS), 7.6(C),
11(S), 6(TS), 14(C)

A single-level index structure for this file is shown in figure 3. If a user query does not specify the security level of the data to be retrieved, then this query must be processed at least at those partitions whose levels are dominated by the level of the search request. This is useful if the search yields data at multiple levels. However, if the relevant records are found in only a small number of the partitions or only at a single level then the effort in searching indices at the other partitions is wasted. This is particularly wasteful if there are a large number of security levels and hence a large number of partitions to search. Note that single-level indices have the advantage that they are easier to maintain and can be maintained locally at each partition.

The alternate approach is to maintain a global multilevel index for the index file *F* as shown in figure 4. It performs well for range queries, but not for single-level queries because the global index is created over all key values.

We try to combine the advantages of both the indexing schemes with our approach. The index structure at the bottom tier consists of a collection of B⁺ trees, one for each single-level partition of the multilevel relation. On top of these single-level indices, we maintain a multilevel B⁺ tree index consisting of selected key values from each of the single-level indices. Unlike the global indexing scheme, the top tier index is really a coarse index; it contains only those values that are necessary to direct a query to the appropriate single-level index. Since we do not want to repeat the search through a single-level index starting from its root, the pointer from the leaf node of the coarse level index points to the leaf node of the relevant single-level index.

Figure 5 shows the proposed index structure for the index file *F*. Single-level queries are evaluated by bypassing the coarse index and going directly to the appropriate single-level index. All other queries are evaluated at the coarse level first and then directed to the leaf nodes of only those single-level indices that may contain the desired key. The following section describes the indexing scheme in greater details.

4 The Indexing Scheme

There are several possible variations of our coarse index structure. Each trades off the amount of storage required by the index structure for the cost of a query. We will describe only one of the index structures, which we choose to call *Coarse Index 1*, in detail; this will give an understanding of the key issues involved in the proposed structure. We will briefly describe a variant, called the *Coarse Index 2*, to give some idea about different coarse index structures and

Key Values: 5 (TS), 7 (TS), 8 (TS), 9.5 (TS), 10 (S), 12 (S), 15 (C), 13 (C), 7.5 (C),
7.3 (C), 20 (S), 21 (S), 22 (S), 9 (TS), 7.6 (C), 11 (S), 6 (TS), 14 (C)

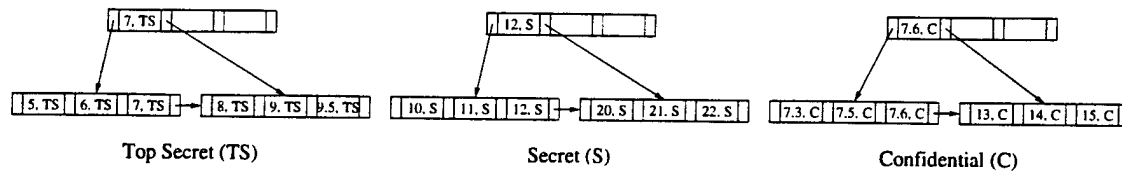


Figure 3. Single-level Index for Each Partition

Key Values: 5 (TS), 7 (TS), 8 (TS), 9.5 (TS), 10 (S), 12 (S), 15 (C), 13 (C), 7.5 (C),
7.3 (C), 20 (S), 21 (S), 22 (S), 9 (TS), 7.6 (C), 11 (S), 6 (TS), 14 (C)

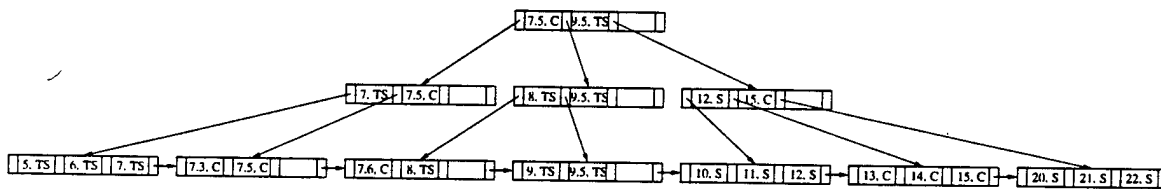


Figure 4. Global Multilevel Index for all Partitions

Key Values: 5 (TS), 7 (TS), 8 (TS), 9.5 (TS), 10 (S), 12 (S), 15 (C), 13 (C), 7.5 (C),
7.3 (C), 20 (S), 21 (S), 22 (S), 9 (TS), 7.6 (C), 11 (S), 6 (TS), 14 (C)

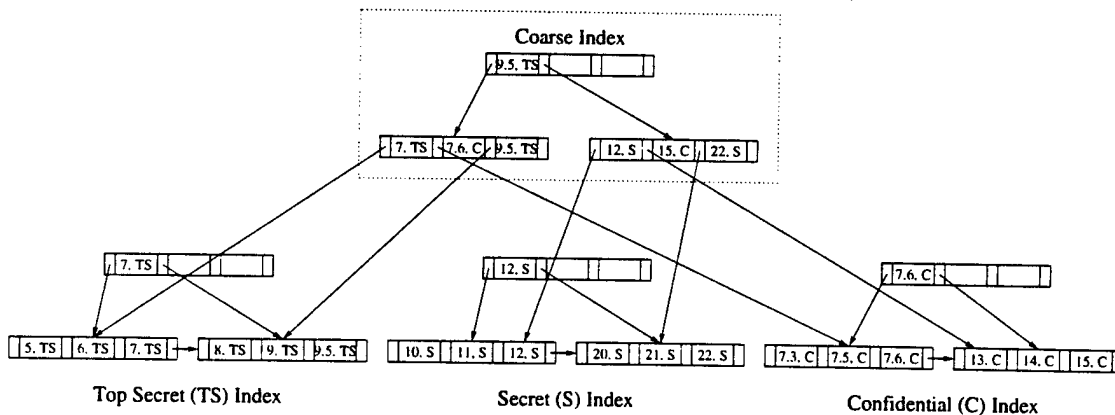


Figure 5. Two-tier Coarse Index for MLS Data

then provide a detailed performance analysis of both index structures. A discussion of other variations of our index structures will be the content of a future work.

Below, when we speak of a key, we assume that it also includes the security label associated with the key.

4.1 Inserting a key in the index

A key K_i is inserted in the coarse index as follows:

1. Search the relevant single-level index structure to find out if the key K_i being inserted is already present. If so, return with an insertion violation error.
2. If K_i can be inserted, then insert it at the relevant single-level index.
3. Determine if K_i and/or any other key K_j from this single-level index need to be inserted at the coarse level index:
 - (a) If, after K_i is inserted at the single-level index, K_i is the largest key in the node into which it was inserted, then K_i needs to be inserted at the coarse level index.
 - (b) If the node of the single-level index in which K_i is to be inserted has to be split into two nodes to accommodate K_i , then the largest keys K_j and K_k in each of the two nodes need to be inserted at the coarse level index. Note that a situation can arise where one of K_j or K_k was the largest key value in the node before it was split. In that case, this key value is already present in the coarse index and need not be re-inserted. Further, one of K_j and K_k may be the same as K_i ; in this case the key will be inserted according to step 3a above.
 - (c) In all other cases there is no need to insert any value at the coarse index and insertion process is complete.
4. If a key K_l (K_i and/or some other key) from step 3 above is being inserted in the coarse index, determine if there is a need to get a key from a single level index other than the one in which K_l belongs (This process is necessary because there is intermingling among the keys):
 - (a) If K_l is to be inserted between keys K_m and K_n in the coarse index, then visit the single-level index pointed to by the pointer P_n (for key K_n), provided this single-level index is different from the one that K_l is in. Note that this pointer points to key values that are less than or equal

to K_n but greater than K_m . Thus, there is a possibility that there are keys in the index pointed to by K_n that are less than K_l . In such a case, get the largest of such keys from the single-level index and insert it along with K_l in the coarse level index.

- (b) If K_l is to be inserted as the smallest key in the coarse index and if the pointer corresponding to the current smallest value in the coarse index points to a single-level index different from the one in which K_l is, then as in step 4a, get the largest key value in this index that is smaller than K_l and insert it in the coarse index along with K_l .
- (c) In all other cases, insert K_l only in the coarse index.
5. Once all the relevant keys have been inserted in the coarse index, along with pointers to the corresponding leaf nodes of the single-level indices containing these keys, the insertion process terminates.

4.2 Deleting a key from the index

We now describe how a key is deleted from the index structure. An important feature to note in this deletion process is that if a key K_i is deleted from the coarse index, we may need to insert another key value K_j from the single-level index containing K_i , such that K_j is the largest value smaller than K_i in the single-level index. If K_j occupies the same position in the coarse index that K_i originally did, then we will just replace K_i with K_j . Otherwise, we may have to insert a second key K_k from a different single-level index as was required in the insertion algorithm.

1. If K_i is not present in the single-level index, then return with a deletion violation error.
2. If key K_i can be deleted, then delete it from the single-level index.
3. If K_i is in the coarse index, then delete K_i from the coarse index. Get a replacement for K_i from the single-level index that K_i was in as follows:
 - (a) If the deletion process results in the merging of two adjacent leaf nodes, then let K_j be the largest key value in the merged node.
 - (b) If the deletion process does not result in any merging of leaf nodes, then let K_j be the largest key smaller than K_i in the node that contained K_i .
 - (c) K_j is K_i 's replacement in the coarse index provided K_j is not already present in the coarse index.

4. If K_j is not already present in the coarse index, insert K_j following the insertion algorithm described earlier. Note that, as in the insertion algorithm, this step may require a second key K_k from some single-level index different from the one containing K_j , to be inserted in the coarse index.
5. This completes the key deletion process.

4.3 Searching for a key in the index

The search process is straightforward and works as follows:

1. If the key K_i being searched for is in the coarse index, locate K_i in a leaf node of the coarse index. Follow the pointer P_i corresponding to K_i to a leaf node of some single-level index. Locate K_i in this leaf node and return with success.
2. If K_i is not in the coarse index, locate the smallest key value K_j in the coarse index that is larger than K_i .
 - (a) Follow the pointer P_j corresponding to K_j to the leaf node of some single-level index containing K_j .
 - (b) Search for K_i in this leaf node. If found, return search successful; otherwise return search unsuccessful.

4.4 A variation of the coarse index structure

Refer to the key insertion procedure described in section 4.1. Note that whenever a leaf node at a single-level index is split, we insert a key value from each of the resulting two nodes in the coarse index. This results in the coarse index having at least one entry in its leaf nodes for each of the leaf nodes of any single-level index. To find a key at a leaf node of a single-level index after the search has traversed the coarse index structure, it requires a search of only one leaf node of the single-level index.

In this variation, henceforth called Coarse Index 2, we no longer require that for every leaf node of a single-level index there be at least one entry in the coarse level index. Instead, for a chain of leaf nodes $N_i, N_j \dots N_p$ in that order, we include some value K_p of N_p at the coarse index and make the pointer of K_p point to leaf node N_i . As a result all key values in $N_i, N_j, \dots N_p$ up to the value K_p are pointed at by the pointer of K_p in the coarse index. To find a key value that is less than or equal to K_p in the single-level index, we follow K_p 's pointer in the coarse index to leaf node N_i , then sequentially search the chain of leaf nodes till we either come to the desired value or get to K_p .

Figure 6.1 shows the Coarse Index 1 and figure 6.2 shows its variant, Coarse Index 2, for the same set of key values.

Note that during insertion or deletion process, we have to find out if there is a value in some other single-level index that needs to be inserted at the coarse index, just as we did for Coarse Index 1. This alternate structure reduces the size of the coarse index considerably at the expense of increased cost of query (that is more number of blocks need to be accessed to get to the key). The complete analysis of both the structures is given in section 5 below.

5 Performance Analysis

In order to assess the suitability of the proposed indexing schemes for specific secure database applications, it is important to determine the costs and benefits of the schemes. We now derive analytical expressions for two chosen performance metrics for the single-level indices, the Global indexing scheme, and the two Coarse Index methods.

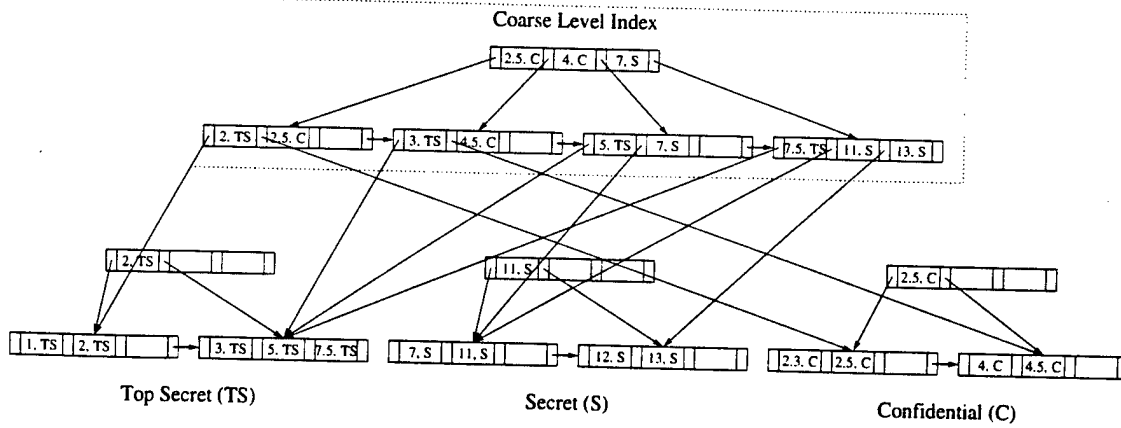
5.1 Analytical model

We model an MLS database system in terms of the following parameters:

- The number of keys (or records) in the system (D)
- The number of distinct security levels (L)
- The distribution of the keys among the different security levels ($R_1 : R_2 : \dots : R_L$)
- The expected number of keys in a given query range (Q) (Here, we assume that query is a read-only operation which specifies a range of keys to be searched.)
- The order of the B^+ trees (P)
- The average fullness factor for the nodes in the B^+ trees (F) (that is, on an average only a fraction F of the entries in any given index tree are filled. Obviously, $1 \geq F \geq 0.5$ by the definition of the B^+ tree.)
- The average size of the clusters (S_1, S_2, \dots, S_L), which is determined by the interleaving of keys of different security levels.

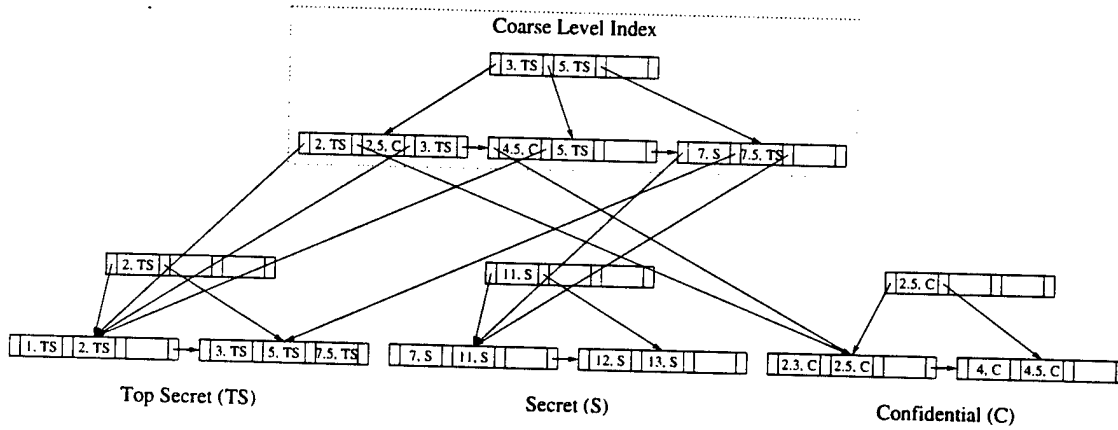
The model parameters are summarized in Table 1. The following example database illustrates the notation. Since it is also used as a base case in the results section, to describe the effect of different parameters on the performance, we refer to it as the base database. The base database has one million keys ($D = 10^6$), with four distinct security levels ($L = 4, U \preceq C \preceq S \preceq TS$), with the keys being distributed in the ratio of 4:3:2:1 among the four classes (where 4, 3, 2, 1 correspond to levels U, C, S, and TS, respectively). The

Key Values: 3 (TS), 7 (S), 5 (TS), 4 (C), 11 (S), 2 (TS), 1 (TS), 2.5 (C), 12 (S), 7.5 (TS), 13 (S), 4.5 (C), 2.3 (C)



6.1: Coarse Index 1

Key Values: 3 (TS), 7 (S), 5 (TS), 4 (C), 11 (S), 2 (TS), 1 (TS), 2.5 (C), 12 (S), 7.5 (TS), 13 (S), 4.5 (C), 2.3 (C)



6.2: Coarse Index 2

Figure 6. The Two Variants of the Coarse Index Structure on the Same Keys

Mnemonic	Description	Base case
D	Total number of keys	10^6
L	Security levels	4 (U, C, S, TS)
R_i	Portion of level i keys	$R_1 = 4, R_2 = 3, R_3 = 2, R_4 = 1$
Q	Average number of keys within a given query range	1000
P	Order of the B^+ tree	10
F	Node fullness factor	1.0
S_i	Average size of level i cluster	$S_1 = 100, S_2 = 75, S_3 = 50, S_4 = 25$

Table 1. Model parameters and Base case

query under consideration covers 1000 keys ($Q = 1000$). The order of the B^+ tree is 10 ($P = 10$) and each node is assumed to be full ($F = 1.0$) at the time the query arrives. The average cluster sizes of the U, C, S, and TS classes are 100, 75, 50, and 25 keys, respectively. For example, on the average, the number of consecutive keys with security level U is 100. The data for the base case is also included in Table 1.

From the basic model, we can derive the following factors which are used in the rest of the analysis:

- Total number of keys of level i , $k_i = D \cdot \frac{R_i}{\sum_{j=1}^L R_j}$
- The average number of clusters of level i , $c_i = k_i/S_i$
- Average number of keys in the query range corresponding to level i , $q_i = Q \cdot \frac{R_i}{\sum_{j=1}^L R_j}$
- Average number of clusters in the query range corresponding to level i , $r_i = q_i/S_i$
- The total number of clusters in the query range, $\tau_\sigma = \sum_{i=1}^L r_i$

Note that we use upper case letters for the basic model parameters and lower case letters for derived parameters.

5.2 Performance metrics

To compare the performance of different index methods, we chose two metrics: size of the index table and the cost of executing a query. These are defined as follows:

1. Size of index table: We measure the size in terms of the number of nodes (often referred to as blocks in literature [3]) of the B^+ tree representing the index structure. Obviously, the smaller the size, the more suited it is for database applications, especially those running on limited memory machines. We denote this metric by α . The metric is computed for the Global index (α_1), the Coarse Index 1 (α_2), the Coarse Index 2 (α_3), and the four single-level indexes ($\alpha_u, \alpha_c, \alpha_s, \alpha_{ts}$).
2. Cost of query execution: During the execution of a range query, an index is searched to access the actual data. Since the number of data blocks to be accessed for executing the query execution is independent of the index method, we have defined the cost of query execution only in terms of the number of nodes (or blocks) of the index structure (B^+ tree) that would be searched (or accessed). This measure is well-suited for comparing the different indexing methods being proposed in this paper. We denote this cost by β . The metric is computed for queries at the four different

levels. It is assumed that a query at a given level needs to access data corresponding to its own level as well as those dominated by it. For example, an S level query accesses data related to U, C, and S levels. Accordingly, the cost of a query at S level includes the cost of accessing indexes at U, C, and S levels. We compute this metric for the Global index (β_1), the Coarse Index 1 (β_2), the Coarse Index 2 (β_3), and the four single-level indexes ($\beta_u, \beta_c, \beta_s, \beta_{ts}$).

5.3 Evaluation of the index size metric

In order to estimate the size of a B^+ tree, we first need to determine its height. Given the order (P), the fullness factor (F), and the number of keys K (or data pointers at the leaf level), the height of a tree (h) can be estimated to be [3]

$$h = \frac{\log\left(\frac{K}{F \cdot (P-1)}\right)}{\log(F \cdot P)} + 1 \quad (1)$$

Here, we assume that while the intermediate nodes of a B^+ tree can hold up to P pointers to lower level nodes, the leaf node only holds up to $(P - 1)$ data pointers as one of the pointers is used for pointing to its right sibling node. Using this equation, the expressions for the single-level indices and the global index can be easily derived by proper substitutions for K . These are summarized in Table 2. The expressions for the coarse indices, however, are more complex. We now derive these expressions.

In the case of Coarse Index 1, the leaf node has one or more pointers to a cluster depending on whether or not a cluster covers one or more nodes at the leaf level of a single index. In other words, if a cluster of keys occupy n (partial or full) nodes at the corresponding index tree's leaf level, then Coarse Index 1 would have n pointers pointing to the n nodes. If n_i denotes the average number of nodes (or blocks) that a cluster of level i occupies in the single-level index at level i , then it can easily be derived as follows.

$$n_i = 1 + \lfloor (S_i - 1)/(F \cdot (P - 1)) \rfloor + \frac{\lfloor S_i - 1 \rfloor \oplus \lfloor F \cdot (P - 1) \rfloor}{(F \cdot (P - 1))} \quad (2)$$

where \oplus represents the modulus operator.

Since there are c_i clusters for level i , Coarse Index 1 would point to $\sum_{i=1}^L (c_i \cdot n_i)$ nodes at the single-level indices. Thus, the height of Coarse Index 1 is given by substituting this term for K in Equation (1).

Since Coarse Index 2 only has one pointer to each of the clusters, it would point to $\sum_{i=1}^L c_i$ nodes at the single

Mnemonic	Expression
h_u	$\frac{\log(K_u/(F \cdot (P-1)))}{\log(F \cdot P)} + 1$
h_c	$\frac{\log(K_c/(F \cdot (P-1)))}{\log(F \cdot P)} + 1$
h_s	$\frac{\log(K_s/(F \cdot (P-1)))}{\log(F \cdot P)} + 1$
h_{ts}	$\frac{\log(K_{ts}/(F \cdot (P-1)))}{\log(F \cdot P)} + 1$
h_1	$\frac{\log(D/(F \cdot (P-1)))}{\log(F \cdot P)} + 1$
h_2	$\frac{\log(\sum_{i=1}^L c_i \cdot n_i / (F \cdot (P-1)))}{\log(F \cdot P)} + 1$
h_3	$\frac{\log(\sum_{i=1}^L c_i / (F \cdot (P-1)))}{\log(F \cdot P)} + 1$

Table 2. Summary of Height Computations

Mnemonic	Expression
α_u	$\frac{(F \cdot P)^{h_u} - 1}{F \cdot P - 1}$
α_c	$\frac{(F \cdot P)^{h_c} - 1}{F \cdot P - 1}$
α_s	$\frac{(F \cdot P)^{h_s} - 1}{F \cdot P - 1}$
α_{ts}	$\frac{(F \cdot P)^{h_{ts}} - 1}{F \cdot P - 1}$
α_1	$\frac{(F \cdot P)^{h_1} - 1}{F \cdot P - 1}$
α_2	$\frac{(F \cdot P)^{h_2} - 1}{F \cdot P - 1}$
α_3	$\frac{(F \cdot P)^{h_3} - 1}{F \cdot P - 1}$

Table 3. Summary of Size computations

indices. Hence, its height is given by substituting this value for K in Equation (1). These are summarized in Table 2.

Once the height h is known, the size of an index table may be computed using the properties of the B⁺ tree as

$$\alpha = \frac{(F \cdot P)^h - 1}{F \cdot P - 1} \quad (3)$$

By substituting the appropriate term for h in Equation (3), we can compute the size metric for the single-level indices ($\alpha_u, \alpha_c, \alpha_s, \alpha_{ts}$), the Global index (α_1), the Coarse Index 1 (α_2), and the Coarse Index 2 (α_3). These are summarized in Table 3.

5.4 Evaluation of the cost of query execution

Let us first consider the case of single-level indices. Clearly, given a U level query, we need to access only the index for U level. Once we reach the leaf node of the index that represents the beginning of the query range, data pointers to other keys may be obtained by traversing the leaf nodes horizontally (using the right sibling links) until

the range is covered. Accordingly, the cost metric β_u is given by

$$\beta_u = h_u + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor \quad (4)$$

For a query of level C, we need to access both keys of type U and C by repeating search on single-level indices of C and U. Accordingly,

$$\beta_c = h_u + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor + h_c + \left\lfloor \frac{q_c}{F \cdot (P-1)} \right\rfloor \quad (5)$$

We can derive similar expressions for β_s and β_{ts} . These are summarized in Table 4.

In the case of Global index, since keys of all levels are present at the leaf, all keys in the query range (i.e., Q keys) need to be searched, for any level of query. Hence,

$$\beta_1 = h_1 + \left\lfloor \frac{Q}{F \cdot (P-1)} \right\rfloor \quad (6)$$

The same β_1 will be applicable for all four types of queries (U, C, S, TS).

The computations of the cost metrics are more complex for Coarse Index 1 and Coarse Index 2. First let us consider the case of Coarse Index 1. For a level j query, we need to find the pointer to the first cluster in the given query range for level j and all levels dominated by j . Hence, we need to search the leaf nodes of the coarse index, starting from the beginning of the query range, going horizontally, until we find the first cluster for each type of the required level. If First(i) represents the average number of nodes to be scanned to find the first cluster of level i , starting from the beginning of query range, then the number of nodes scanned in Coarse Index 1, before all the required first cluster pointers are obtained, is given by $\max_{v_i, i \leq j} \text{First}(i)$. In addition, since the corresponding single-level indices have also to be scanned horizontally, the cost of execution is given by

$$\beta_{2,j} = h_2 - 1 + \max_{1 \leq i \leq j} \text{First}(i) + \sum_{1 \leq i \leq j} \left\lfloor \frac{q_i}{F \cdot (P-1)} \right\rfloor \quad (7)$$

First(i) may be computed by using probabilistic arguments. (We omit the details here.) It is given as

$$\text{First}(i) = \frac{r_i}{r_\sigma} + \sum_{k=1}^{r_\sigma - 2r_i} \left(\prod_{l=0}^{k-1} \left(1 - \frac{r_i}{r_\sigma - l} \right) \right) \cdot \frac{r_i}{r_\sigma - k} \cdot \left\lfloor \frac{k \cdot t_i}{F \cdot (P-1)} \right\rfloor \quad (8)$$

Mnemonic	Expression
β_u	$h_u + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor$
β_c	$h_u + h_c + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_c}{F \cdot (P-1)} \right\rfloor$
β_s	$h_u + h_c + h_s + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_c}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_s}{F \cdot (P-1)} \right\rfloor$
β_{ts}	$h_u + h_c + h_s + h_{ts} + \left\lfloor \frac{q_u}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_c}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_s}{F \cdot (P-1)} \right\rfloor + \left\lfloor \frac{q_{ts}}{F \cdot (P-1)} \right\rfloor$
β_1	$h_1 + \left\lfloor \frac{Q}{F \cdot (P-1)} \right\rfloor$
$\beta_{2,j}$	$h_2 - 1 + \max_{1 \leq i \leq j} \text{First}(i) + \sum_{1 \leq i \leq j} \left\lfloor \frac{q_i}{F \cdot (P-1)} \right\rfloor$
$\beta_{3,j}$	$h_3 - 1 + \max_{1 \leq i \leq j} \text{First}'(i) + \sum_{1 \leq i \leq j} \left\lfloor \frac{q_i}{F \cdot (P-1)} \right\rfloor$

Table 4. Summary of Cost of Query Executions

where t_i is the average size of clusters in the query range excluding cluster i . ($t_i = \sum_{l=1, l \neq i}^L (r_l/r_\sigma) \cdot S_l$)

The computations for Coarse Index 2 are quite similar except that this index carries only one pointer per cluster. Thus, the expression for $\beta_{3,j}$ is given as follows.

$$\beta_{3,j} = h_3 - 1 + \max_{1 \leq i \leq j} \text{First}'(i) + \sum_{1 \leq i \leq j} \left\lfloor \frac{q_i}{F \cdot (P-1)} \right\rfloor \quad (9)$$

where

$$\text{First}'(i) = \frac{r_i}{r_\sigma} + \sum_{k=1}^{r_\sigma - 2r_i} \left(\prod_{l=0}^{k-1} \left(1 - \frac{r_i}{r_\sigma - l} \right) \right) \cdot \frac{r_i}{r_\sigma - k} \cdot \left\lfloor \frac{k}{F \cdot (P-1)} \right\rfloor \quad (10)$$

6 Results

To determine the effect of different system parameters on the two performance metrics (α and β) with different indexing methods, we have evaluated the metrics under different configurations. Our evaluation methodology is as follows.

1. The base case for the evaluation is as described in Table 1.
2. Under each of the configurations, we evaluate the index size (α) for the four single-level indices (U, C, S, TS), the Global index (GI), the Coarse Index 1 (CI1) and Coarse Index 2 (CI2). The cost of query metric (β) is evaluated for all four types of queries (U, C, S, TS). For each type, we evaluate the metric when only the single-level index for that type is available, as well as for the Global index, and the two Coarse indices.

3. To evaluate the effect of cluster size on the metrics, we vary the cluster size (S) in the base model. In the current runs, we keep the ratio of the cluster sizes constant at 4:3:2:1 as in the base model but varied the constant factor from 1 through 10^6 . So the cluster sizes were varied from the set $\langle 4, 3, 2, 1 \rangle$ to $\langle 4 \cdot 10^6, 3 \cdot 10^6, 2 \cdot 10^6, 10^6 \rangle$. Due to space limitations, only a subset of the results are presented in figures 7 and 11.
4. To evaluate the effect of key ratio size on the metrics, we vary the key ratio (R_i 's) in the base model. We evaluated the metrics under a different set of ratios including 1:1:1:1, 10:1:1:1, 100:1:1:1, 1000:1:1:1, ..., 1:1:1:1000. Due to space limitations, the results are not plotted, but the summary of the observations is presented below.
5. To evaluate the effect of the tree order (P), we vary the order from 2 through 500. The results are summarized in figures 8 and 12.
6. To evaluate the effect of number of keys (D), we experimented with D values from 100 to 10^8 . Portions of the results are summarized in figures 9 and 13.
7. To evaluate the effect of the fullness factor (F), we changed values of F from 0.7 through 1.0. The results are summarized in figures 10 and 14.
8. To determine the query range effect, Q was varied from 1 through 10^6 . The observations are summarized below.

Following is a summary of our observations regarding the behavior of different indexing methods under different model parameters.

- Index size (α): As mentioned above, index size is a very important metric since it determines the overall

performance of the system. If the index size is small enough that it can fit in the fast memory of the system, then quick turnaround times may be achieved for query executions. Following is a summary of our analysis illustrating the effect of different system parameters on the size of various index methods.

- (i) Effect of Cluster size (S_i): Since the single-level indices and the Global index have pointers to individual data and not to clusters, this parameter has no effect on their sizes (i.e., α_s). On the other hand, it has significant impact on the sizes of Coarse Index 1 and Coarse Index 2 (see figure 7). As the size of the clusters increase, the number of clusters decrease. This means that the number of elements pointed to by Coarse Index 2 also decrease. For example, when the cluster sizes of the four classes were 40, 30, 20, 10, respectively, Coarse Index 2 requires 4938 nodes (or blocks). But when the size of each cluster is ten times (i.e., 400, 300, 200, 100), the size decreases to 494, a tenth of the previous. Thus the Coarse Index 2's size is inversely proportional to the cluster size.

Since Coarse Index 1 points to all nodes that a cluster of data keys occupy at a single-level index, the relationship between its size (α_2) and the cluster size is not so straightforward. In general, increase in cluster size would decrease the index size but not as dramatically as in the case of Coarse Index 2. For example, in our experiments, when the cluster sizes were changed from 40, 30, 20, 10 to 400, 300, 200, 100 respectively, for the four levels, the index size reduced from 20987 nodes to 14444 nodes.

- (ii) Effect of Key ratio (R_i): Since the sizes of single-level indices directly depend on the number of keys they have to point to, the key ratio has an effect on individual index sizes. However, the sum of their sizes remains essentially unchanged. Thus while the sizes of single-level indices were 30864 each when the ratio was 1:1:1:1, it changed to 49382, 37036, 24961, 12345 respectively, when the ratios were changed to 4:3:2:1. Observe that the difference in the sum of sizes in the two cases is not significant. Obviously, the size of the Global index is unaffected by the ratios since it has keys of all levels. Similarly, this factor does not have significant impact on the sizes of Coarse Index 1 and Coarse Index 2.
- (iii) Effect of Tree order (P): Since the order determines the maximum number of elements in

a node of an index tree, the size of the tree decreases with the increasing order (see figure 8). While the decrease is considerable for smaller values of tree order, the effect is not so dramatic at higher values of tree order. For example, while the size of Coarse Index 2 reduced from 32000 to 7100 when order is increased from 2 to 4, it only reduced from 330 to 160 when the order is doubled from 50 to 100. The performance shows similar trends for all index types.

- (iv) Effect of Number of Keys (D): The sizes of all indices grow linearly with the number of keys (see figure 9). For example, when the number of keys is increased from 10^4 to 10^6 , the size of Coarse Index increased from 20 to 2000. Similarly, the size of Global index grew from 1235 to 123500 for the same changes in the number of keys.
- (v) Effect of Fullness factor (F): An increase in fullness factor implies that the same tree size can accommodate more number of keys. However, since B⁺ tree requires that each node be at least half-full, F 's effect on the size is not as significant as other factors. However, it can be noticed that the size decreases, somewhat linearly, with the fullness factor (see figure 10). For example, as the fullness factor is increased from 0.7 to 0.8 and then to 0.9, the size of Coarse Index 2 changed from 2963 to 2540 and then to 2222.
- (vi) Effect of Query Range (Q): Obviously, there is no effect of query range on the index size.
- Cost of Query (β): This metric, which indicates the number of node (or block) accesses required to execute a query, influences the execution time of the query. Obviously, for a lower turnaround time, we prefer as few nodes to be accessed as possible. Following is a summary of our analysis illustrating the effect of model parameters on this metric. Due to space limitations we have included the results for query types U and TS in figures 11-14.
 - (i) Effect of Cluster Size: Since the structure of the single-level indices and the Global index are unaffected by clustering, the query cost is also unchanged under these cases (figure 11). In the case of Coarse Index 1, given any query range, the index guarantees a pointer to the single-level index that begins the range. Hence, except in some special cases where the cluster size is not a multiple of the node order (or $F \cdot P$), this metric is not sensitive to the cluster size.

The situation is quite different with Coarse Index 2. Here, since the index guarantees a pointer only to the beginning of a cluster in the single-level index, it is sensitive to the cluster range. For example, if a cluster with a size of 1000 starts from key 100 and ends with key 1099 (here, for simplicity we assumed that all keys from 100 to 1099 exist), then a query requiring the range of 100-5000 is pointed to the same block in the single-level index as the one with a query range of 1098-6000. Hence, the second query has to search several additional nodes at the leaf of a single-level index before it encounters the block with 1098 from where the search starts. Larger the cluster size, larger is such overhead. In the example cases that we studied, for cluster sizes of up to 100, the performance of both index methods was comparable. Beyond cluster size of 300, Coarse Index 2 becomes expensive.

- (ii) Effect of Key Ratio (R_i): While this has some effect on the cost of single-level index accesses and coarse index accesses, it has no effect on the Global index. The query cost using Coarse Index 1 is affected to a small extent. For example, when the key ratio is changed from 10:1:1:1 to 1000:1:1:1, the query cost for U with Coarse Index 1 increased from 90 to 115. Similarly, for TS query, it increased from 155 to 170. For the same changes, the single-level index access cost increased from 90 to 115. In fact, the increase in the Coarse Index 1 is to be mainly attributed to the increase in single-level index access only. For the same changes, the cost with Coarse Index 2 increased from 95 to 120.
- (iii) Effect of Tree Order (P): The tree order has similar effects on the query cost as it has on the index size—while the rate of reduction in the cost is exponential at the smaller values of P , it is not so large for larger values of P (see figure 12). For example, for a U query, while the cost for Coarse Index 2 reduced from 470 to 160 when order is increased from 2 to 4, it only reduced from 13 to 8 when the order is doubled from 50 to 100. The performance shows similar trends for all index types and all query types.
- (iv) Effect of Number of Keys (D): Since all indices grow logarithmically with the number of keys, the cost also grows only logarithmically with the number of keys (see figure 13). For example, when the number of keys is increased from 10^4 to 10^6 , the cost of U query with Coarse Index 2 increased from 53 to 55. Similar changes

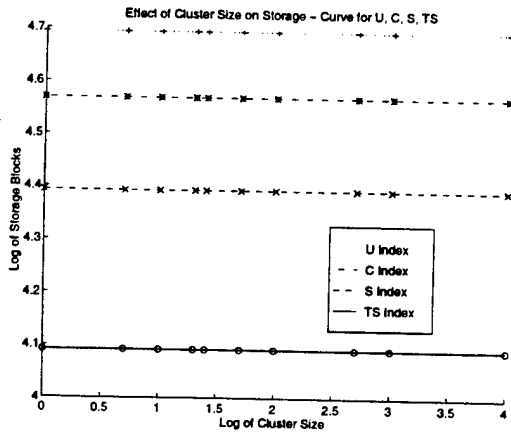
with Coarse Index 1 resulted in an increase of cost from 57 to 59.

- (v) Effect of Fullness factor (F): As in the case of the index size, the fullness factor results in increased block accesses while descending the index trees during searching as well as while performing a horizontal search at the leaf nodes (see figure 14). Thus, the query cost increases as the fullness factor decreases. For example, when the Fullness factor is increased from 0.7 to 0.9, the cost of U query with Coarse Index 2 decreased from 77 to 60. Similar changes with Coarse Index 1 resulted in a decrease of cost from 83 to 65.
- (vi) Effect of Query Range (Q): Since a horizontal scanning of nodes involving all the keys in a given range is required at the leaf level of the indices, this factor has an effect on the query cost. Since each node of the tree holds $F \cdot P$ pointers or 10 in our case, the increase is logarithmic (with base 10). For example, when the query range is increased from 100 to 10000, the cost of U query with Coarse Index 2 increased from 15 to 455. The same change with Coarse Index 1 resulted in an increase in cost from 10 to 460. With Global index, the cost increased from 17 to 1117.

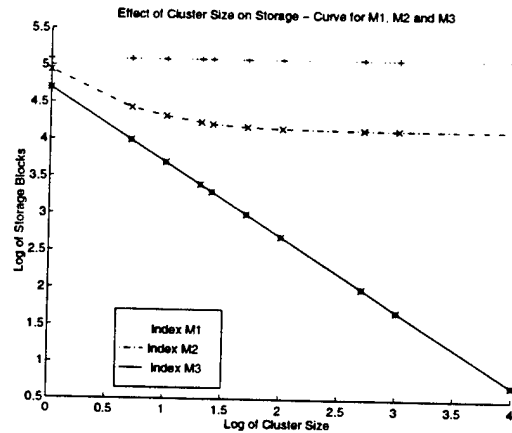
7 Conclusion

In this paper, we have introduced two types of coarse indexing schemes — Coarse Index 1 and Coarse Index 2 — in the context of MLS databases, primarily with the intent of improving query response time and reducing the size of indices. The functionality of these indices is explained in terms of key clusters where a cluster corresponds to a sequence of consecutive (in the global order) keys with the same security level. In Coarse Index 1, there are pointers to each of the leaf nodes of the single-level indices covering the keys in the cluster. On the other hand, Coarse Index 2 carries one pointer per cluster. The proposed indexing schemes require a two-phase search technique while executing a MLS query. In the first phase, a coarse index is searched to determine a position in the single-level index (corresponding to one of the single-level databases). In the second phase, this positional information is used to carry out search on the index (or indices) of single-level databases.

We have developed detailed algorithms for each of these schemes. We have then compared the performance of the proposed indices with single-level indices as well as a Global multilevel index that contains keys of all security levels. B^+ trees are used to implement indices and

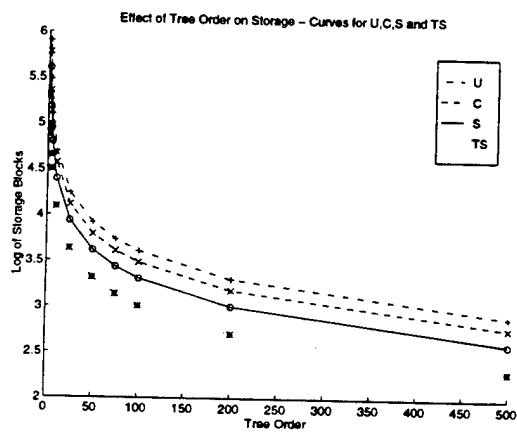


7.1: Graph for U, C, S and TS

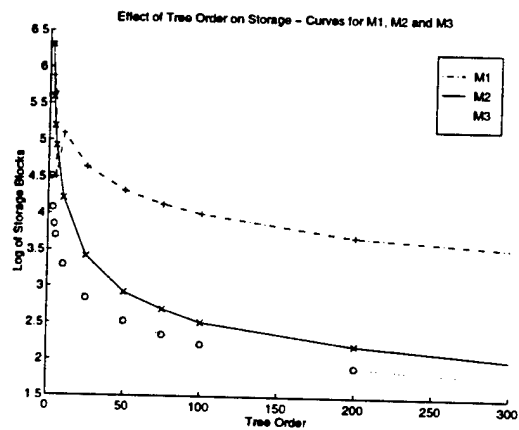


7.2: Graph for GI, CI1 and CI2

Figure 7. Effect of Cluster Size on Index Size

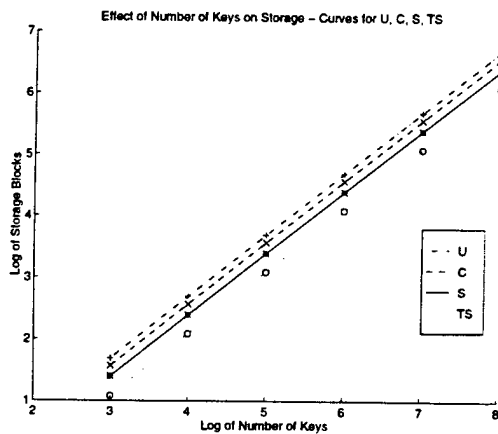


8.1: Graph for U, C, S and TS

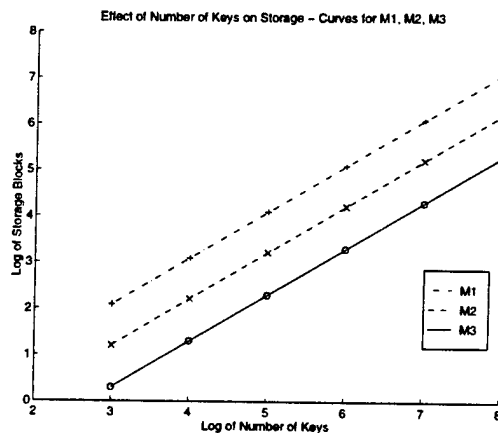


8.2: Graph for GI, CI1 and CI2

Figure 8. Effect of Tree Order on Index Size

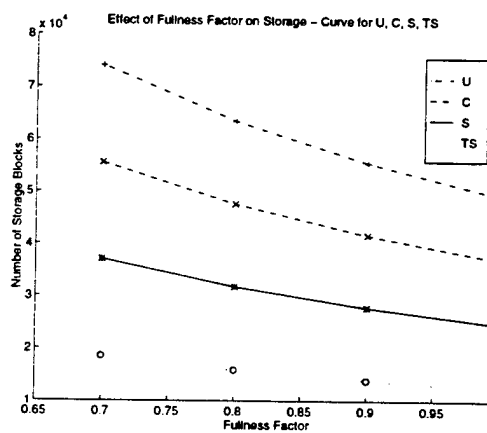


9.1: Graph for U, C, S and TS

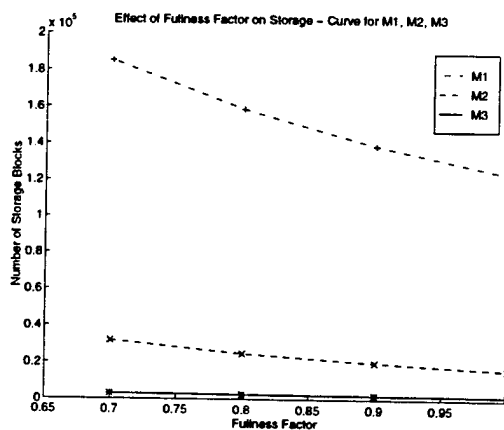


9.2: Graph for GI, CI1 and CI2

Figure 9. Effect of Number of Keys on Index Size

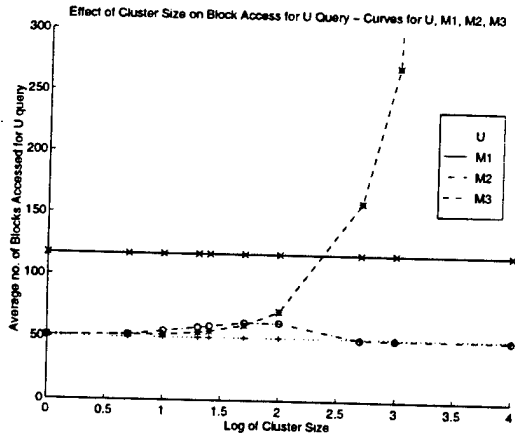


10.1: Graph for U, C, S and TS

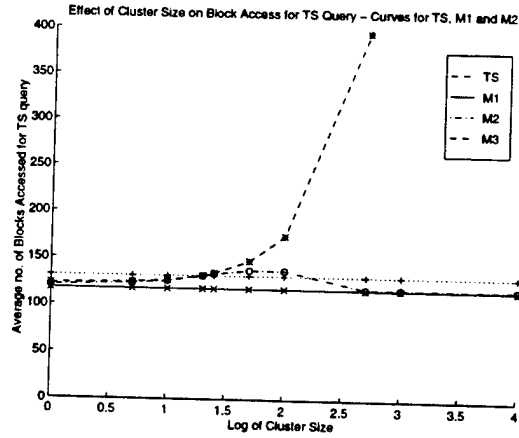


10.2: Graph for GI, CI1 and CI2

Figure 10. Effect of Fullness Factor on Index Size

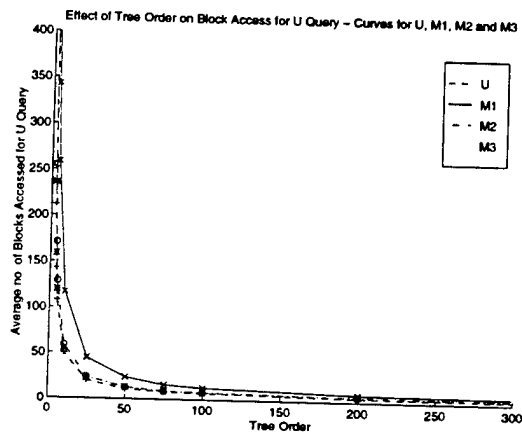


11.1: Graph for U, GI, CI1 and CI2

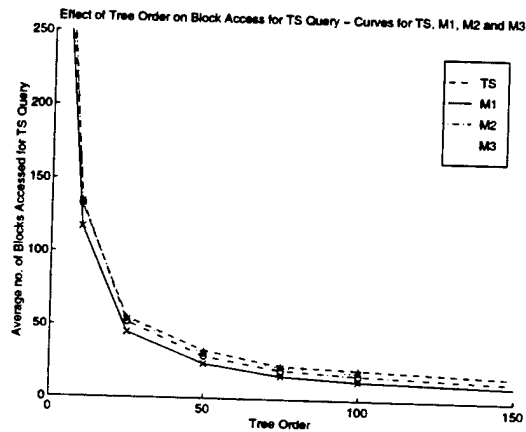


11.2: Graph for TS, GI, CI1 and CI2

Figure 11. Effect of Cluster Size on Cost of Query

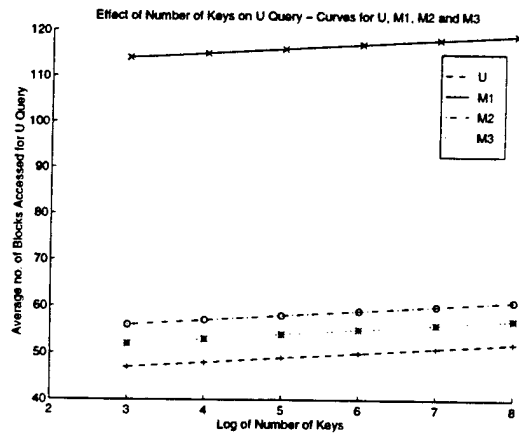


12.1: Graph for U, GI, CI1 and CI2

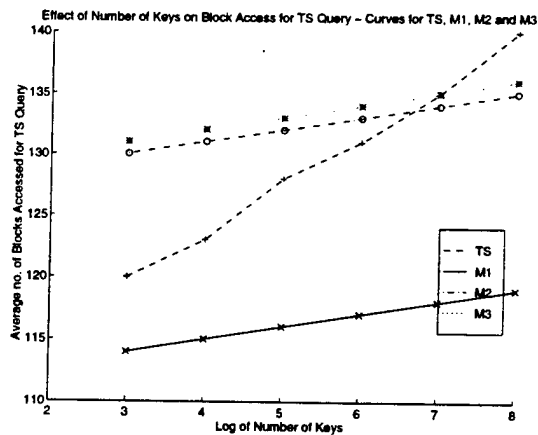


12.2: Graph for TS,GI, CI1 and CI2

Figure 12. Effect of Tree Order on Cost of Query

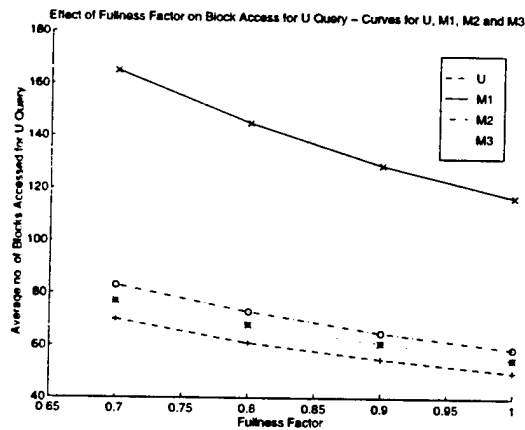


13.1: Graph for U, GI, CI1 and CI2

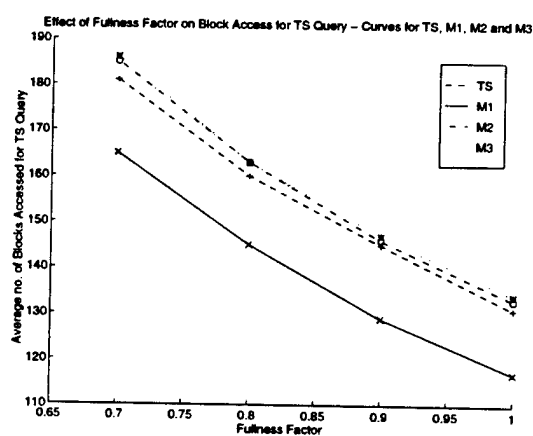


13.2: Graph for TS, GI, CI1 and CI2

Figure 13. Effect of Number of Keys on Cost of Query



14.1: Graph for U, GI, CI1 and CI2



14.2: Graph for TS, GI, CI1 and CI2

Figure 14. Effect of Fullness Factor on Cost of Query

two performance metrics—size of index and cost of query execution—are defined for the comparisons. We developed an analytical model for the indices where each index is modeled by seven parameters—number of keys, number of security levels, distribution of keys among levels, size of a query range, order of the B⁺ tree, the fullness factor of the B⁺ tree, and the average size of clusters.

We have then run several experiments to compare the performance of the indexing schemes under different sets of parameter values. We conclude that while both the coarse index methods result in reduced index size, Coarse Index 2 is found to be much more effective in achieving this objective. The cost of Coarse Index 2, however, is noticeably higher in query executions with cluster sizes of 200 or higher. The cost of query execution with Coarse Index 1 remained unaffected by the cluster sizes. Except for the sensitivity to the cluster size, Coarse Index 2 retained its supremacy over Coarse Index 1 and over the Global index. The single-level indices, seem to have marginal advantage over Coarse Index 2 in the cost of query metric.

With the encouraging results from this study, we propose to look at other coarse indexing schemes. Especially, we propose to look at schemes in which a coarse index points to a security level that a cluster belongs to, and at schemes in which the coarse index points to some intermediate node of a single index where a cluster begins.

Finally, we have assumed that the coarse index is trusted. It would be interesting to investigate a kernelized implementation of the coarse index.

References

- [1] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.
- [2] R. Bayer and M. Schkolnick. Concurrency of Operations on B-Trees. *Acta Informatica*, 9:1–21, 1977.
- [3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, Reading, MA, second edition, 1994.
- [4] Informix Software. *Informix-OnLine/Secure Security Features User's Guide*. Menlo Park, CA, 1993.
- [5] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.
- [6] H. F. Korth and A. Silberschatz. *Database System Concepts, Second Edition*. McGraw-Hill, New York, 1991.
- [7] P. Lehman and S. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transaction on Database Systems*, 6(4), dec 1981.
- [8] G. Luef and G. Pernul. Supporting Range Queries in Multilevel-Secure Databases. In C. E. Landwehr and S. Jajodia, editors, *Database Security, V: Status and Prospects*, pages 117–130. Elsevier Science Publishers B.V. (North-Holland), 1992.
- [9] Oracle Corporation. *Trusted Oracle7 Server Administrator's Guide, Release 7.2*. Redwood City, CA, 1996.

- [10] Sybase, Inc. *Sybase Secure SQL Server Security Administration Guide*. Emeryville, CA, 1993.

Replication Does Survive Information Warfare Attacks*

John McDermott

Center for High Assurance Computer Systems

Naval Research Laboratory, Washington, DC 20375-5537

mcdermott@itd.nrl.navy.mil

Introduction

Ammann, Jajodia, McCollum, and Blaustein define *information warfare* as the introduction of incorrect data intended to hinder the operation of applications that depend on the database [2]. In describing their approach to surviving these kinds of attacks on databases, imply that replication is not useful in dealing with information warfare attacks. In this paper we present results to the contrary, i.e. replication can be used (carefully) to both detect and survive information warfare attacks, on a practical basis.

McDermott and Goldschlag [4, 5] define *storage jamming* as "malicious but surreptitious modification of stored data, to reduce its quality. The person initiating the storage jamming does not receive any direct benefit. Instead, the goal is more indirect, such as deteriorating the position of a competitor." This is essentially the same as information warfare, and we adopt the latter term. To provide context, Amman et al. specifically do not consider Trojan horses within the database system (called *internal jammers* [5]), but instead consider a wide range of attacks other than Trojan horses. Both groups agree that Trojan horses are more effective attackers, since they can access data which the human attacker cannot. McDermott et al. show how to detect sophisticated attacks by Trojan horses inside the database system but do not address recovery or continued operation. Amman et al. do not address detection. Instead, they show not only how to assess damage after an attack, but also how to continue operation with partially damaged data. This paper seeks to show how replication can be used to not only detect attacks, but to assess damage and continue operation, thus surviving information warfare attacks.

* This work was supported by ONR. Any opinions, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views, policies, or decisions of the Office of Naval Research or the Department of Defense.

In this paper, we borrow some terms from Ammann et al. and refer to data damaged by an attack as either *red data* or *off-red data*. *Red data* is unsafe to use; *off-red* has been damaged, but may be used. *Green data* is valid and has not been damaged. We also use *red* and *green* to describe values that are to be stored as data, by the attacker and the defender respectively.

Replication as a Defense: Detection

Replication in general is problematic in an information warfare context. Under many replication approaches, *red* data can be replicated automatically and precisely to many locations. However replication works as a defense if we use (*one-copy serializable*) *logical replication over distinct database systems*. Many replication algorithms copy data values from the source data item to its replicas. However, logical replication copies the command that caused the source data item to change. The command is executed at each replica's site and, because of one-copy serializability, results in the same new value for the replica. If we assume a distinct provenance (defined in the next section) for the database system software at each site, then the Trojan horse will not be replicated at all sites. An attack must compromise multiple, possibly heterogeneous, host programs, an unlikely event in practical systems. Even if the attackers can succeed at every site, the attack still may fail. If the Trojan horses are not able to deliberately malfunction in a one-copy serializable fashion, their *red* values will diverge. This can be ensured by restricting communication between the sites to just the protocols needed to carry out the authorized replication. So we can expect a scheme using n replicas to detect up to $n-1$ cooperating Trojan horses and possibly detect an n -Trojan horse attack.

Detection is simple in the replication defense. There is a detection process at each source or replica site. Following changes to protected data, the process at the source site computes a checksum over the changed data and sends it to each replica site, along with the identification of the change. After the logical update is performed at a replica site, the detection process at the replica site computes its own checksum and compares it to the checksum transmitted by the source site detection process. If there is disagreement, there is a problem. Checksums are not essential to the approach and are merely used to facilitate efficient comparison. The granularity of the comparisons or checks is a tradeoff between speed and storage. Comparisons over individual data items allow quicker response to attacks but take more storage to perform. We also do not need to check every change, since the insertion of bogus data at some sites will ultimately diverge the copies.

If we establish checksums over our entire database, detection can be effective against both external jammers and internal jammers. (External

jammers attack files outside their host application, e.g. a Trojan horse hosted by an Oracle database system that attacks Mathematica files is an external jammer.) Indistinguishability [4] comes for free, without our being able to define or verify it.

Aircraft	Pilot	Tanker	Refuel
Sword 1	Bong	Coke 1	2000
Sword 2	McGuire	Coke 2	2000
Axe 1	O'Hare	Coke 2	3000

Table 1 . The *refueling* relation

Tanker	Fuel Available
Coke 1	3000
Coke 2	9000

Table 2. The *tanker* relation

We use the aircraft refueling example of Ammann et al. Suppose we have a relational database with two relations *refueling* and *tanker*. Both relations are replicated at three sites: *cactus*, *yucca*, and *sorrel*. The checksums for *refueling* and *tanker* are r_1 and t_1 , respectively. At site *yucca*, a command is issued to **"update refueling set tanker = 'Coke 2' where aircraft = 'Sword 1'"**, but there is a Trojan horse in the database system at *yucca* that sets *tanker* = 'Coke 1' where *aircraft* = 'Sword 2'. The new (incorrect) checksum for refueling is r_2 , which is sent to sites *cactus* and *sorrel*, along with the command **"update refueling set tanker = 'Coke 2' where aircraft = 'Sword 1'."** At both *cactus* and *sorrel*, the requested change is made to *refueling*, but the detection processes compute a different checksum r_3 , because the result of correctly executing the command is different. Either detection process can now report a problem because $r_2 \neq r_3$. Notice that, in this example, we must compute checksums for relation *tanker*, because the Trojan horse may have modified the *tanker* relation while performing **"update refueling set tanker = 'Coke 2' where aircraft = 'Sword 1'"** correctly. If the Trojan horse had been at site *cactus* the attack would still be detected by the difference in checksums. (We defer an

example of damage assessment and continued operation until later in the paper.)

It is important that the detection process be separate from the database system. Otherwise, it might be possible for the Trojan horse to send checksums for *green* values to other sites while writing *red* values at its own site. Furthermore, the detection process must be trusted, i.e. it must be high assurance software that is protected from tampering. Finally, we are assuming that the Trojan horse can be located and removed using existing system administration tools.

Although this approach is reminiscent of Byzantine generals approaches, we do not recommend extending it to carry out a similar automatic agreement protocol. The foremost reason for this is that our approach is intended to work with shrink-wrapped general purpose software. It is unlikely that software vendors will modify their products to carry out the cryptographically protected voting protocols needed to reach Byzantine agreement. A less important reason is that use of such protocols for every update would seriously impact the performance of most database systems. At present it is more expedient to detect the attacks and then remove the Trojan horse.

Distinct Provenance

Software that is created, delivered, installed, and maintained by distinct sets of people has a distinct provenance. Distinction can be forced to many levels by a variety of techniques. Since multibase techniques allow replication over heterogeneous systems, the database systems at each site can be different, even if they are shrink-wrapped general purpose software packages.¹ Shrink-wrapped general purpose software packages (e.g. the database system software) can be purchased through blind buys, which simulates distinct provenance. Applications, site-specific software, macros, etc. can be developed using clean room techniques. In a clean room approach, developers provide inspected source code to each site. The source code is converted to executable form (e.g. compiled and linked, converted to p-code) and installed at the operational sites by personnel distinct to each site. Maintenance and administration can likewise be separated site-wise by clean room techniques. Our notion of distinct provenance is not the same as *n*-version programming. We are not trying to tolerate inadvertent bugs but to deny an attacker access to multiple sites. The expectation is that we have now forced would-be attackers to

¹ Introduction of heterogeneity may require the use of trusted mapping functions that are assured to map the logical update commands in a way that preserves checksums.

compromise multiple host programs in very sophisticated ways. A practical n -Trojan horse attack can only succeed if all n Trojan horses can maintain one-copy serializability over all changes to their red data and internal states. Since the successful Trojan horses cannot be replicas of each other, this is problematic for the attacker. If we assume, say a software development team, has m members who understand the software² then the n -Trojan horse attack reduces to an mn -person manual attack.

In theory, a distinct provenance is possible. In practice, some software may have commonality. Some distinct software will either have been developed with the same tools or be based on the same packages. This raises the question of Trojan-horse-writing Trojan horses [3]. Fortunately, a would-be attacker introducing an attack via widely-used software faces a significant problem. The problem is that the Trojan horse's lifetime is now likely to be expended against systems other than the target. The Trojan horse will trigger on sites that are not the intended target. The attacker must now arrange to turn off the Trojan horse in systems that are not targets or risk premature discovery of the attack. Attacks via automatic data input systems face the same problem. More red data must be created, and not all of it will be put in the target database. This increases the chance of someone detecting the attack by inspection of the data.

Manual Attacks

Logical replication is clearly a problem for Trojan-horse-based attacks because those attacks function by "disobeying" the commands given to the software. So we have frustrated the most effective means of attack. But what about less effective manual attacks? Manual attacks are carried out by giving malicious commands to the database system. We can deal with manual attacks in one of two ways: 1) by incorporating an n -person rule [1], or 2) by incorporating transaction control expressions [7]. An n -person rule requires n humans outside the system to agree to a change to the database. Transaction control expressions are a more general form of this concept. They require multiple users to agree to specific conditions defined on specific steps of a transaction. In either case, we assume that data manipulation commands are legitimate unless all n persons can collude.

We also note that in newer automated systems, the amount of manual input to a database is less than in the past. For example, tanker aircraft may have on-board software that automatically reports the amount of fuel

² The point here is that m is not as large as the entire team, but in a well-managed properly assured software development program greater than unity. Under poorly-managed, low-assurance development, we are not sure any defense is possible.

carried. Refueling assignments to aircraft may be calculated by a decision aid program. This appears to be just moving the problem around, but in fact it reduces the opportunities for manual attacks. Information warfare attacks via automatic data input suffer from the same weakness as Trojan horses written into mass-produced software (see below).

Application Attacks, Interface Attacks, etc.

Careful readers will question whether application programs can be abused to simulate the advantages of manual attacks while avoiding transaction control expressions or n -person rules. If an application outside the database contains the attacking Trojan horse, it can submit commands to insert bogus values and the database system will replicate the bogus commands as though they were manual commands. Fortunately, this type of attack is frustrated by replicating the application software, i.e. defensive replication is not limited to database systems. The same type of attack can be made via any software (we hope not via hardware or firmware!) that lies between a system's input devices and its output devices. Careful replication of these components will suffice to detect such attacks just as the basic database attacks are detected. Our approach does have trouble with the connections between a system and its I/O peripherals. When we finally reach the devices that lie at the boundaries of our system, things become unclear. In a theoretical sense, we can define the problem away by saying that attacks that modify data as it is being put in or out are not information warfare attacks. In a practical sense, we would have to limit our replication to components that handle the most critical data.

Replication as a Defense: Damage Assessment

Ammann et al. introduced the important concept of *damage markings*. Damage markings are attributes that indicate the degree of damage that has been assessed upon a particular data item. We also adopt damage markings and use their scheme.

Leaving other considerations such as system errors aside, when a check fails and we detect an attack, we should expect that either the source database system has been compromised or the replicas that failed the check have been compromised. All systems participating in the defense should be alerted. We assume that database administrators and support teams will eventually locate the Trojan horse and remove it. Data items relating to the change that failed the check should all be marked *red*, even though some will in fact be *green*. The correct values can be determined by manual inspection, by simple majority vote over all copies of a data item. Correct copies of the data are then marked *green*. Copier transactions can use the *green* replicas to repair damaged copies of data items.

Suppose we look at damage assessment in our refueling example. *Red* markings on the copies of *refueling* at sites *cactus* and *sorrel* can be changed to *green* by the damage assessment transaction. Note that markings for relation *tanker* need not be changed at any point during this attack. When the detection processes at sites *cactus* and *sorrel* detect the attack, all tuples of the *refueling* relation can be temporarily marked *red*, on the basis of the text of the command that failed the checksum. Damage assessment in this case can be accomplished by majority vote, which allows us to identify the second tuple of *yucca*'s copy of *refueling* to be damaged. Tables 3 and 4 indicate the state of the database after damage assessment, with *red* data shaded.

Aircraft	Pilot	Tanker	Refuel
Sword 1	Bong	Coke 2	2000
Sword 2	McGuire	Coke 1	2000
Axe 1	O'Hare	Coke 2	3000

Table 3 . Marking the damaged *refueling* relation at site *yucca*

Aircraft	Pilot	Tanker	Refuel
Sword 1	Bong	Coke 2	2000
Sword 2	McGuire	Coke 2	2000
Axe 1	O'Hare	Coke 2	3000

Table 4 . Marking the damaged *refueling* relation at sites *cactus* and *sorrel*

Replication as a Defense: Continued Operation

The use of logical replication may allow us to disconnect compromised systems until the Trojan horse can be disabled. If an uncompromised site can act as a source site, it can take over from a compromised source. Replica sites that do not originate data are also easily disconnected.

A more complex approach would logically "disconnect" compromised data items (e.g. classes or relations)

Defensive Partition

If the compromised site is not the source of the data or there is an alternate source site, then the replicated database system can be partitioned into a damaged and an undamaged component. The partition can take place after damage assessment and could be decided on the basis of an agreement algorithm, just like the damage assessment. Any compromised sites are placed in the damaged component. The sites in the undamaged component can continue to operate normally. Sites in the damaged component would only be allowed to submit read requests to sites in the undamaged partition.

Single-Source Data

If the replication is done with only one source site, and that site is compromised, then we conjecture that we can still use a modified version of the continued operation protocol of Ammann et al. Their protocol uses transactions that distinguish between inputs, outputs, pure reads, updates (read and write), and blind writes, as well as insert and delete. Our modifications for continued operation under single-source replication are:

1. We do not use the *off-green* marking. Correct³ values of every data item will be available for repair of every detected attack. We decide at database design time whether a data item will be marked *red* or *off-red* during damage assessment.
2. We do not use the Coincidental Damage Deletion rule because it may allow incorrect deletion of *off-red* data that will ultimately be marked *green* by a damage assessment algorithm⁴. We do use the other rules listed below. Notice that the remaining rules have been modified to incorporate replication.
 - a. Confinement: A normal transaction T that attempts to read, update, blind write, or delete a data item accesses any available *green* copy. If no *green* copy is available, the normal transaction attempts to read an *off-red* copy. If no *off-red* copy is available, then T rolls back. A normal transaction may not create *red* data.

³ but not necessarily up-to-date

⁴ We assume that the presence of correct copies of the data makes it likely that this repair will take place in a relatively short period of time.

- b. Propagation: If a transaction T reads data marked *off-red*, then any output by T is marked *off-red*. Transaction T may not update, blindly write, or delete data for which a *green* copy is available. Transaction T may not create *off-red* data.
- c. Coincidental Repair of Off-Red Data: If a transaction reads only *green* data then any *off-red* data item it writes blindly is marked *green*.

X

4. We simplify the definition of consistency by leaving out the acceptable but not necessarily consistent integrity constraints, giving us the following definition of integrity

- a. For each integrity constraint $i \in I$, where I references exclusively *green* data, i holds.
 - b. For each integrity constraint $i \in I$, where I references data items x_1, \dots, x_n that are not *green*, there exist values for x_1, \dots, x_n such that i is satisfiable.
4. All data is initially marked *green*. Markings are changed by a damage assessment algorithm, from *green* to *red* or *off-red*, iff the data is damaged. Damage assessment transactions do not change any data, but correctly identify valid copies of data items that have damaged replicas. Markings are changed by a copier transaction repairing damage, from *red* or *off-red* to *green*.

A normal (i.e. not a copier, attacker, or damage assessor) transaction T preserves consistency if, given a consistent and all *green* state S_1 , T produces a consistent all *green* state S_2 .

We now pose a theorem analogous to one of Ammann et al., namely

Theorem

Suppose a consistency preserving normal transaction T follows the modified continued operation protocol defined above, and S_1 is a consistent state of the possibly damaged database. Then state S_2 , the state resulting from the application of T to S_1 , is consistent.

Proof:

1. The Confinement rule prohibits transaction T from accessing any *red* data. Transaction T cannot violate I by reading or writing *red* data.

2. The Propagation rule allows T to cause *green* data to become *off-red*. Consider an integrity constraint i . If some copy of a data item x referenced in i is green in state S_1 and becomes off-red in state S_2 as a result of transaction T 's actions, then there must be values for a copy of data item x that satisfy i , that is value of x in S_1 . (Even though green replicas of x are often available, all copies of x may be temporarily marked *red* or *off-red* by a damage assessment transaction.) A transaction that reads *off-red* data under the Propagation rule cannot modify *green* data without causing it to become off-red. For this reason, *green* data in the new state S_2 must also have been marked *green* in state S_1 . Integrity constraints in I are satisfied in S_2 .
3. The Coincidental Repair of Off-Red Data rule allows a normal transaction T to change the marking of an *off-red* data item x to *green* when writing blindly, if T only reads *green* data. Since transaction T is consistency preserving, the values it writes when changing x satisfy I in S_2 .

To return to our running example: for continued operation we could at first not use any data from relation *refueling*. After damage assessment, the relations would be marked as shown by Tables 3 and 4. We could read and modify the copies of *refueling* at sites *cactus* and *sorrel* even though the damage was not repaired. Ultimately, a copier transaction could repair the "Sword 2" tuple of *yucca*'s copy of *refueling*, by copying correct values from either *cactus* or *sorrel*.

Stored Procedures

Stored procedures are widely used in current databases. Their impact on storage jamming is problematic. First of all, the stored procedure mechanism is an ideal tool for building efficient, sophisticated jammers. Stored procedures also make good hiding places. On the other hand, all but the most sophisticated jamming attacks against stored procedures are probably too risky for the attacker. Plausible values for passive data items are easy to generate, either by arithmetic or by copying components (e.g. fields). Applying simple arithmetic to the text of a stored procedure does not necessarily result in a plausible, valid program text. Copying substrings of a program text into the target procedure may result in a valid program, but probably not a plausible one.

Predictability is also an issue for the attacker. The modified procedure may exhibit spectacular behavior that immediately reveals the Trojan horse. Programs that can automatically generate valid program texts that also implement specific algorithms are still in the research stage. They are also relatively large, i.e. on the order of general purpose database sys-

tem software, so they would be difficult for the attacker to hide. Inserting bogus code into multiple stored procedures could result in a combinatorial explosion of bad data that would also reveal the attack. It is possible that future research in automatic code generation could make it possible to build a small malicious program that surreptitiously modifies stored procedures.

Stored procedures require extra care on the part of the defenders. They must be replicated, but with distinct provenance. They should not be automatically copied or translated to the various sites, but should be reviewed outside the database systems and then installed manually. If distinct provenance is maintained, replication should be an effective means of defending against jamming of stored procedures.

Conclusions

Before presenting our conclusions, we would like to discuss some key assumptions we are making, so that the application of our results will be clear:

1. We assume that some malicious software can be introduced into most systems during their lifetime. We assume that introducing specific malicious software into multiple sites is problematic and cannot be done repeatedly or at will.
2. We assume malicious software or users can be removed from a system soon after they are detected. This is not always so in real life, but it is possible in systems following best practice.
3. The following software components must be trusted: the detection process that computes, compares, and transmits checksums, any mapping functions used to translate logical updates to site-specific languages, damage assessment voting or agreement algorithms, and copier transactions used to repair damage. To warrant this trust they must be correct, unbypassable, and tamper-proof. We assume sufficient access control, audit, and cryptographic systems to make this be so.

Replication via logical updates is a viable defense that allows detection of, damage assessment after and continued operation during information warfare attacks. With n replicas, logical replication is effective in detecting automatic (Trojan horse) attacks involving less than mn person collusion, where m is the number of members of a software development or maintenance team. With n -person data entry, logical replication is effective in detecting manual attacks involving less than n -person collusion. With transaction control expressions, the likelihood of successful manual attack is even less.

Detection of an attack by logical replication results in an undamaged copy of the target data, at either the source or the replica site. A simple majority of undamaged copies is sufficient to identify the correct values. Even if there is no majority, possession of the text of the offending command will allow (admittedly tedious) identification of the correct value.

The continued operation protocol of Amman, Jajodia, McCollum, and Blaustein can be used to operate a replicated database system prior to identification of the correct values. Once the correct values have been identified, the database can operate from either *green* copies or our modification of the original continued operation protocol. The existence of identifiably correct copies makes it possible to intentionally partition the damaged database system, thus isolating the offending subsystem.

At present we are prototyping proof-of-concept software for a replicated architecture defense. Our target system is SQL Server running on Windows NT. Future work should include more sophisticated continued operation protocols that account for both communication failures and site failures. Specific damage assessment algorithms, accompanied by improved damage marking schemes may be beneficial to improved recovery from attacks.

Acknowledgements

We would like to thank David Goldschlag, Carl Landwehr, and Robert Gelinas for their contributions to this paper. The thoughtful comments of the anonymous referees have improved the paper considerably.

References

1. AVIZIENIS, A. The methodology of n-version programming. In *Software Fault Tolerance* (M. LYU ed.), John Wiley, 1995.
2. AMMANN, P. JAJODIA, S., MCCOLLUM, C., and BLAUSTEIN, B. Surviving information warfare attacks on databases. In *Proceedings of IEEE Computer Society Symposium on Security and Privacy*, Oakland, California, May, 1997.
3. McDERMOTT, J. A technique for removing an important class of Trojan horses from high-order languages. In *Proceedings 11th National Computer Security Conference*, Baltimore, 1988, 114-117.
4. McDERMOTT, J. and GOLDSCHLAG, D. Storage jamming. In *Database Security IX: Status and Prospects* (D.SPOONER, S. DEMURJIAN, and J. DOBSON, eds.), Chapman and Hall, London, 1996.

5. McDERMOTT, J. and GOLDSCHLAG, D. Towards a model of storage jamming. In *Proceedings IEEE Computer Security Foundations Workshop*, p. 176-185, Kenmare, Ireland, June 1996.
6. McDERMOTT, J. Practical defenses against storage jamming. 20th National Information Systems Security Conference, Baltimore, MD, October 1997.
7. SANDHU, R. Separation of duties in computerized information systems. In *Database Security IV: Status and Prospects* (J. JAJODIA and C. LANDWEHR, eds.), North-Holland, 179-189.

Priority-driven Secure Multiversion Locking Protocol for Real-Time Secure Database Systems*

Chanjung Park, Seog Park and Sang H. Son[†]

Department of Computer Science
Sogang University
Seoul, 121-742, Korea
{cjpark,spark}@dmlab.sogang.ac.kr

[†]Department of Computer Science
University of Virginia
Charlottesville, VA22903
son@cs.virginia.edu

Abstract

Database systems for real-time applications must satisfy timing constraints associated with transactions. Typically a timing constraint is expressed in the form of a deadline and is represented by a priority to be used by schedulers. In many real-time applications, security is another important requirement, since the system maintains sensitive information to be shared by multiple users with different levels of security clearance. As more advanced database systems are being used in applications that need to support timeliness while managing sensitive information, protocols that satisfy both requirements need to be developed.

In this paper, we propose a new priority-driven secure multiversion locking(PSMVL) protocol for real-time secure database systems. The schedules produced by PSMVL are proven to be one-copy serializable. We have also shown that the protocol eliminates covert channels and ensures that high priority transactions are neither delayed nor aborted by low priority transactions. The details of the protocol, including the compatibility matrix and the version selection algorithm are presented. Several examples to illustrate the behavior of the protocol are provided, along with performance comparisons with other protocols.

1 Introduction

A multilevel secure database management system (MLS/DBMS) is a transaction processing system which is shared by users with more than one clearance level and which contains data of more than one classification level[8][9]. In order to control all the accesses to the database, mandatory access control(MAC) mechanisms are adopted in MLS/DBMS. With MAC mechanisms, the sensitive data is protected by permitting access by only the users whose

security levels are higher than or equal to the levels of data. In order for MLS/DBMS to be correct, it has to meet security requirements in addition to satisfying logical data consistency. The most important requirements for multilevel security are the elimination of covert channels between transactions of different levels and the starvations of high-level transactions[8][9].

In principle, MLS database systems should be used for any system that contains sensitive data[13]. In real-time database management systems(RTDBMS), transactions have explicit timing constraints such as deadlines[12]. The time criticalness(priority) of a transaction usually derives from both its timeliness requirement and its importance. RTDBMS must satisfy timing constraints associated with transactions and maintain data consistency. There are increasing needs for supporting applications which have timing constraints while managing sensitive data in advanced database systems. To support such applications, we must integrate real-time transaction processing techniques into MLS/DBMS, namely MLS/RT DBMS[14]. Since MLS/RT DBMS needs to support both MLS and RT requirements, it is easy to see that protocols for MLS/RT DBMS could be more complicated than those for MLS/DBMS or RTDBMS. There are several ongoing research projects on concurrency control protocols for RTDBMS and MLS/DBMS. However, the protocols for MLS/RT DBMS are rarely presented. Recently, SRT-2PL(Secure Real-Time Two Phase Locking) protocol[11] for MLS/RT DBMS was proposed. The protocol tried to satisfy two requirements, but there still exists the priority inversion problem¹.

In this paper, we propose a priority-driven secure multiversion locking protocol, called PSMVL, for MLS/RT DBMSs. The proposed protocol ensures that high-priority

*This work was supported in part by IITA(Institute of Information Technology Assessment)(Project NO. 96054-IT2).

¹A priority inversion occurs when a high-priority transaction is delayed by a low-priority transaction. It is not desirable in real-time database systems.

transactions are not blocked due to low-priority transactions for timing constraints, while low-level transactions are not interfered with by high-level transactions to avoid covert channels. The protocols based on multiversions require more amount of storage than those based on a single version. However, the proposed protocol is based on multiversion scheme for some reasons. First, disk prices have come down dramatically, the disk space needed to store multiple versions is cheaper. Second, the concurrency control protocol which maintains a single version of each data item, such as 2PL-HP[2] and OPT-wait[7] or OPT-sacrifice[7], cannot avoid the starvations of high level transactions, because low level transactions should neither be delayed nor be aborted to prevent covert channels. And the protocol cannot eliminate the starvations of low priority transactions, since low priority transactions can be delayed or aborted by high priority transactions. Third, the protocols which maintain two versions of each of data items can partly resolve the above starvation problems. However, due to the limited number of versions, when a high priority transaction at a high level conflicts with a low priority transaction at a low level on the same data item, the protocols sacrifice one of the requirements. Therefore, multiversion scheme is considered appropriate to satisfy all the requirements for MLS/RT DBMSs. In addition, since our protocol is based on multiversion, late transactions can read the old version of each data and thus, it can increase the degree of concurrency. We have shown that the histories² produced by the protocol are one-copy serializable³.

The rest of the paper is organized as follows. In Section 2, we present the security model of this paper and then introduce the features of transactions in RTDBMS. In Section 3, we classify the transactions according to their characteristics to discuss the conflicting natures of the requirements, and present the PSMVL protocol and the version selection algorithm. In Section 4, we present an example to illustrate the behavior of the protocol. In Section 5, we prove the correctness of the protocol and show that it ensures serializability, security requirements, and no priority inversion. After the performance results of the protocol are presented in Section 6, we conclude the paper in Section 7.

² A history indicates the order in which the operations of transactions are executed relative to others.

³ When we prove the correctness of a multiversion concurrency control protocol, we must show that the multiversion(MV) histories generated by the protocol are one-copy serializable. An MV history H is *one-copy serializable* if its committed projection, $C(H)$, is equivalent to a 1-serial MV history, where $C(H)$ is the history obtained from H by deleting all operations that do not belong to committed transactions in H [6]. A serial MV history H is *1-serial* if for all i, j , and some data item x , if T_i reads x from T_j , then $i = j$, or T_j is the last transaction preceding T_i that writes into any version of x .

2 Background

Let us the security level of a transaction T is denoted by $L(T)$ and the security level of a data item x is denoted by $L(x)$. When transactions access data items, the following security policies are adopted as ours.

- (1) *Simple security property* for read operations[4]: A transaction T is allowed to read a data item x if and only if $L(T) \geq L(x)$.
- (2) *Restricted star property*(\star -property) for write operations: A transaction T is allowed to write into a data item x if and only if $L(T) = L(x)$.

The above two restrictions are intended that sensitive data are protected by permitting only the users whose security levels are higher than or equal to the levels of data. In other words, read/write operations at the same level and read operations at the lower level(read-down) are allowed.

There are three MLS properties: value, delay, and recovery secure properties[8]. To present the definitions, the *purge* function is introduced. Given a schedule S and a security level SL , $purge(S, SL)$ is the function that removes all operations from S , whose level is greater than SL . For an input schedule S , the output schedule S' is *value secure* if $purge(S', SL)$ is view equivalent to the output schedule produced for $purge(S, SL)$. For an input schedule S and an output schedule S' , a schedule is *delay secure* if for each level SL in S , any operation o_i in $purge(S, SL)$ is delayed in the output schedule produced for $purge(S, SL)$ if and only if it is delayed in $purge(S', SL)$. A schedule is *recovery secure* if a set of transactions, T , is in a deadlock state when every transaction in T is waiting for an event that can only be caused by other transactions in T .

A key feature of RTDBMS is that each transaction has timing constraints [1]. The concept of *value function* is adopted as the way of representing the timing constraints of real-time transactions. For each transaction, the output of the corresponding value function expresses the amount of profit that can be obtained by the completion of the transaction before its deadline. Since it is more advantageous to the system for transactions with the largest values to be completed before their deadlines, *high-priority* is given to the transactions that have a large output value. At run time, high-priority transactions should precede low-priority transactions.

In this paper, we adopt the priority assignment policy proposed in [15]. In that policy, each transaction has an initial priority and a start-timestamp. The initial priority of a transaction indicates the criticality of the transaction. The practical priority consists of the initial priority and the start-timestamp.

3 The PSMVL protocol

In this section, we examine the conflicts between real-time and security requirements, and then present the protocols and related rules.

3.1 Motivations

Let T_i and T_j be transactions in a conflicting mode and let $P(T_i)$ and $L(T_i)$ be the priority of T_i and the security level of T_i , respectively. Then, there are three possible cases for the priorities of these transactions: (1) $P(T_i) = P(T_j)$, (2) $P(T_i) > P(T_j)$, (3) $P(T_i) < P(T_j)$. Since (2) and (3) are symmetric, without loss of generality we can consider just one, say (2). Therefore, we can assume that $P(T_i)$ is higher or equal to $P(T_j)$. In addition, there are three cases for the security levels of these transactions: (1) $L(T_i) = L(T_j)$, (2) $L(T_i) > L(T_j)$, (3) $L(T_i) < L(T_j)$.

Let P_{High} , P_{Low} , and P_{Eq} be the priorities and $P_{High} > P_{Low}$. Let L_{High} , L_{Low} , and L_{Eq} be the security levels and $L_{High} > L_{Low}$. In Table 1, L_{Eq} is used in the case that two transactions have the same security level. Table 1 shows all possible combinations of priority and security level pairs between T_i and T_j .

Trans. Cases	T_i		T_j	
	Priority	Security level	Priority	Security level
1	P_{Eq}	L_{Eq}	P_{Eq}	L_{Eq}
2	P_{Eq}	L_{Low}	P_{Eq}	L_{High}
3	P_{Eq}	L_{High}	P_{Eq}	L_{Low}
4	P_{High}	L_{Eq}	P_{Low}	L_{Eq}
5	P_{High}	L_{Low}	P_{Low}	L_{High}
6	P_{High}	L_{High}	P_{Low}	L_{Low}

Table 1: Priorities and Security levels between T_i and T_j

In the first case, the priorities and the security levels of the two transactions are the same. Therefore, the only concern is ensuring serializability. Security requirements and timing constraints can be ignored in this case. In the second and the third cases, the priorities of the two transactions are the same. Hence, timing requirements can be ignored and the low level transaction should not be delayed by the high level transaction. In the fourth case, the security levels of the two transactions are the same. The transactions must be scheduled so that they meet the timing constraints as well as the logical consistency. In this case, any protocol based on the multiversion scheme for RTDBMS can be used. In the fifth case, since $P(T_i) > P(T_j)$, T_i must be followed by T_j in order to prevent priority inversion. In addition, T_i can neither be delayed nor aborted by T_j to avoid covert channels. Both requirements can be satisfied by having T_i precede T_j .

The problem occurs in the sixth case where $P(T_i) > P(T_j)$ and $L(T_i) > L(T_j)$. Since $P(T_i) > P(T_j)$, T_i should

not be blocked by T_j . On the other hand, T_j cannot be blocked by T_i in order to avoid covert channels. We call this kind of conflict *HH/LL-conflict*. It is a conflict between a high security level transaction with high-priority and a low security level transaction with low-priority. Ideally, T_i should precede T_j because of their priorities. We resolve the HH/LL-conflicts by using the proposed PSMVL protocol.

3.2 The compatibility matrix

Like the multiversion two phase locking(MV2PL) protocol[5], PSMVL has three types of locks: read, write, and certify locks. The locks are governed by the compatibility matrix in Figure 1. Since no conflict occurs between read/write or write/write operations, the certify locks are needed in order to get the correct synchronization among transactions. The scheduler adopting PSMVL protocol acquires read and write locks before processing read and write operations, respectively. When a transaction has terminated and is about to commit, the scheduler converts all of the transaction's write locks into certify locks.

We only consider the cases where lock requesters and lock holders have different security levels. As already mentioned, P_{High} and P_{Low} are priorities such that $P_{High} > P_{Low}$ while L_{High} and L_{Low} are security levels such that $L_{High} > L_{Low}$. Let $T_H(P_H, L_H)$ be a lock holder with priority P_H and security level L_H . Similarly, let $T_R(P_R, L_R)$ be a lock requester with priority P_R and security level L_R . The conflicts between all operations of lower level transactions and write or certify operations of higher level transactions cannot occur because of our security policy. There are four cases based on both the priorities and the security levels.

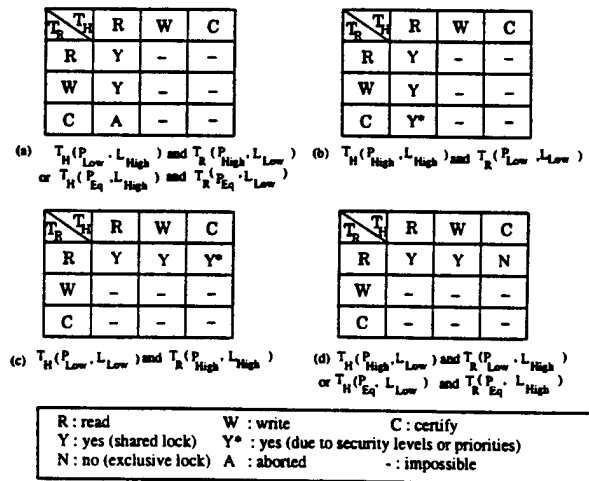


Figure 1: The compatibility matrix for PSMVL

In Figure 1 (a), $P(T_R) \geq P(T_H)$ and $L(T_R) < L(T_H)$. For priority and security reasons, T_R cannot be blocked. Therefore, T_H should be aborted. The abortion of T_H helps that T_H can read more recent data without violating any requirements.

In Figure 1 (b), $P(T_R) < P(T_H)$ and $L(T_R) < L(T_H)$. Under the BLP model, this situation can occur only when the operation of T_R is write while the operation of T_H is read-down. In this case, T_H cannot be blocked in order to avoid priority inversion and T_R cannot be delayed for security reasons. T_H is inserted into $HH\text{-list}(x)$ where x is the data item T_R writes. $HH\text{-list}(x)$ is used for keeping the orderings of priorities and is defined in the next section in detail. In Figure 1 (c), $P(T_R) > P(T_H)$ and $L(T_R) > L(T_H)$. In this case, T_R cannot be blocked because of its priority and T_H should not be delayed in order to avoid covert channels. Since T_R and T_H cannot be blocked, T_R can share a lock and T_R is inserted into $HH\text{-list}(x)$ where x is the data item T_H writes. In Figure 1 (d), $P(T_R) \leq P(T_H)$ and $L(T_R) > L(T_H)$. T_R cannot block T_H because of its priority and security level. Therefore, T_R is blocked until T_H commits.

3.3 Version management

Concurrency control protocols based on multiversion locking use 2PL for write/write synchronization and version selection for read/write synchronization [6]. When a transaction is about to choose a version of a data item, the most recent commit version is commonly used. However, since there exist HH/LL-conflicts in the environment where RT and MLS requirements should be considered together, in order to resolve HH/LL-conflicts, the certify operation of a low level transaction with low priority in (b) of Figure 1 is permitted, and the read operation of a high level transaction with high priority is permitted in (c) of Figure 1. Therefore, additional rules for version selection are required.

Figure 2 shows the data structures for versions used in this protocol. In an MLS/RT DBMS, each data item has its own security level and each transaction has a priority and a security level. *Data_itemT*, *VersionT*, and *Read_downT* are data types. *Data_itemT* is a data type for each of data items, while *VersionT* is used for storing the versions of each data item. And *Read_downT* is a data type for the data items which are read by high level transactions.

In an MLS/RT DBMS, each data item has its own security level and each transaction has a priority and a security level. Each data item contains two fields: *level* and *version*. The *level* represents the security level of a data item and it must be trusted.

The *version* is the field for a version, and contains a *timestamp*, a *value*, a *hhlptr*, and a *vlink*. The *vlink* is the pointer to the next version. The *hhlptr* is a pointer

Data_itemT:

Field name	Type	Description
level	level	The security level of a data item
version	VersionT	A version of a data item

VersionT:

Field name	Type	Description
timestamp	time	The creation time of a version
value	value	The value of a version.
hhlptr	Read_downT pointer	The pointer that points to a node which contains The number of active higher security level transactions that read down the version.
vlink	VersionT pointer	The pointer that points to the next version of data item.

Read_downT:

Field name	Type	Description
level	level	A security level of one or more active transactions that read down the version.
count	integer	The number of active transactions that read down the version for a given security level.
clink	Read_downT pointer	The pointer that points to the next node typed Read_downT(self-referential).

Figure 2: Data structures

that resolves HH/LL-conflicts and maintains the number of higher security level transactions that read down the version. Let T_j and T_k be two transactions with the HH/LL-conflict relationship on some data item x . Let $P(T_j) > P(T_k)$ and $L(T_j) > L(T_k) = L(x)$. In our security policy, this situation can occur when T_j executes a read down operation while T_k executes a write operation. The *hhlptr* is the field of x_i that T_j reads, i.e., x_i is the old committed version of x available to T_j . Since the high priority transaction T_j can read old versions of x , T_j need not be blocked until the lower level transaction T_k writes. The *hhlptr* must be trusted. The *hhlptr* consists of three fields: *level*, *count*, and *clink*. The *level* and the *count* represent the security level of transactions that read down the version in HH/LL-conflicting mode and the number of the transactions, respectively. The *clink* is the pointer that points to the next node for lower level transactions in HH/LL-conflicting mode.

When a HH/LL conflict occurs, the following procedure, called *HH/LL-procedure*, is needed for maintaining the version information. HH/LL-procedure is presented in Algorithm 1.

For each data item x , we maintain a list of transactions, denoted by $HH\text{-list}(x)$, in order to preserve the orderings of priorities. The $HH\text{-list}(x)$ is a list of higher priority and higher security level transactions that are active when another transaction executes a write operation. $HH\text{-list}(x)$ can be obtained by a lock table⁴. If a transaction T_j writes

⁴A lock table contains the information that which transactions have

x at t_{now} , while another transaction T_j with higher priority and higher level than T_i is reading x , then T_i is inserted in $HH\text{-list}(x)$. This insertion means that HH/LL -conflict can occur in the future because $P(T_i) > P(T_j)$. When a transaction reads a data item, HH -list is used to select the appropriate version according to its priority in the version selection algorithm.

/* When a transaction T_j selects the version x_i of x , the following steps are required. Discussion of the version selection algorithm will be provided in a later section. In the algorithm, 'u->v' denotes that v is a member of u . */

```
<Type declaration>
xi      : Data_itemT
new.node: Read_downT
FIND    : boolean type whose value is either TRUE
or FALSE.

if (xi->hhlptr is null) then
  create a node new;
  new->count = 1;
  new->level = L(Tj);
  xi->hhlptr = new;
else
  node = xi->hhlptr;
  FIND = FALSE;
  while (node is not null) do
    if (node->level is L(Tj)) then
      node->count = node->count + 1;
      FIND = TRUE;
      break the loop;
    end if
  end while
  if (FIND is FALSE) then
    create a node new;
    new->count = 1;
    new->level = L(Tj);
    append new to xi->hhlptr;
  endif
endif
```

Algorithm 1: *HH/LL-procedure*

Let $T_i(P_i, L_i)$ be a transaction with priority P_i and security level L_i . Assume that there are two transactions $T_1(P_1, L_1)$ and $T_2(P_2, L_2)$ where $P_1 > P_2$ and $L_1 > L_2$. Let the levels of two data items x and y be L_2 . Suppose that T_1 and T_2 execute as shown in Figure 3 (a). For each data item x , $HH\text{-list}(x)$ is initially empty. At time 2, $HH\text{-list}(x) = \{T_1\}$. At time 4, HH/LL -conflict occurs and the versions for x are as shown in Figure 3 (b). At time 5, T_1 commits and the count for x_0 becomes 0. Then the version x_0 can be deleted.

Three different operations can be performed on a version: creation, deletion, and selection. Since there is no write/write conflict in the PSMVL protocol, a new version can be created without delay. Because of HH/LL -conflicts, two or more old versions must be stored. The version that is older than the latest committed version can be deleted when there exist no high level transactions that read that version. Let $T_i(P_i, L_i)$ be a transaction with priority P_i

locks on some data items.

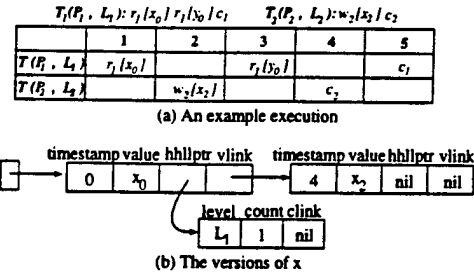


Figure 3: *The versions of data item x*

and security level L_i . When $T_i(P_i, L_i)$ is about to read a data item x , the version selection algorithm (Algorithm 2) selects an appropriate version of x for T_i .

/* When a transaction T_i reads a data item x , this procedure specifies the steps for selecting the right version of x . In the algorithm, 'u->v' denotes that v is a member of u . */

```
<Type declaration>
x      : Data_itemT
hhlptr : Read_downT
FIND    : boolean type whose value is
either TRUE or FALSE.

if  $T_i$  has a write lock on  $x$ , then  $T_i$  must read
the version  $T_i$  writes;
else
  let hhlptr(a variable) be the node linked
  to the hhlptr of the first version of  $x$ .

  FIND = FALSE;
  for (all versions of  $x$ ) do
    if (hhlptr is null) then
      let hhlptr be the node linked to
      the hhlptr of the next version of  $x$ .
    else
      /* hhlptr is not null, i.e., some higher
      level transactions already read down
      the version */
      for (all nodes linked to the hhlptr) do
        find a node such that the level of the
        node is less than or equal to  $L(T_i)$ .
        if (there exists such a node) then
          FIND = TRUE;
          mark the version as  $x_j$ ;
          break the loop;
        end if
      end do
    end if
  end do
  if (FIND is TRUE) then
    return  $x_j$ ;
  else
    find the version such that it is the latest
    committed version of  $x$  before  $T_i$  reads the
    read operation;
    return the version;
  end if
end if
```

Algorithm 2: *Version selection algorithm*

3.4 The protocol

Before we present our protocol, we define the following timestamp assignment rules for our protocol that are used in our protocol to select appropriate versions.

```

/* Let x be a data item and T, T' be transactions.
In addition, let R, W, and C be read, write, and
certification operations, respectively. */
case 1: T requests a read operation on x
  if (x is locked with W or C) then
    if (any lock-holder has a higher or equal
        priority) then
      if (any lock-holder has a lower or equal
          security level) then
        block lock-requester ;
      end if
    else
      /* lock holder has a lower priority */
      Let T' be the transaction that has
        lower priority than that of T;
      if (L(T') is lower than L(T)) HH/LL-procedure();
      else if (L(T') is the same as L(T)) then
        for (all T' which has C on x) do
          convert the lock from C to W;
        end do
      end if
    end if
  end if
  grant a read lock to T;

case 2: T requests a write operation on x
  Let T' be any transaction that already has a lock
  on x(lock holder).
  if (P(T') is higher than or equal to P(T)) then
    if (L(T') > L(T)) HH/LL-procedure();
  else /* P(T') < P(T) */
    if (L(T) = L(T')) then
      T' is blocked by T;
      wait;
    end if
  end if
  grant a write lock to T;

case 3: T requests a certify operation on x
  Let T' be any transaction that already has a lock
  on x(lock holder);
  if (P(T') > P(T)) then
    if (L(T') = L(T)) then
      the request is rejected and blocked;
    else if (L(T') > L(T)) HH/LL-procedure();
    end if
  else if (P(T') = P(T)) then
    if (L(T') = L(T)) then
      if (T' holds a read or certify lock) the
        request is rejected;
      else if (L(T') > L(T)) HH/LL-procedure();
      end if
    else /* P(T') < P(T) */
      if (L(T') = L(T)) then
        if (T' holds a certify lock) then
          for (all T' which has C on x) do
            convert the lock from C to W;
          end do
        end if
      end if
      else if (L(T') < L(T)) HH/LL-procedure();
    end if
  end if
  grant C on x;

```

Algorithm 3: The protocol

First, for each data item x , timestamp $TS(x)$ is given to x when x is created. Second, for a read-only transaction T_i , the starting timestamp, $S_TS(T_i)$ is assigned. For an update transaction T_j , both the starting timestamp, $S_TS(T_j)$

and the committing timestamp, $C_TS(T_j)$ are assigned. We assume that the system guarantees the uniqueness of each timestamp.

The compatibility matrix shown in Figure 1 is the basis for the PSMVL protocol. When transactions have the same priority and the same security level, it behaves similarly to the MV2PL protocol.

3.5 The properties of PSMVL protocol

Let H be a history over a set of transactions $\{T_0, T_1, \dots, T_n\}$ produced by PSMVL. Then, H must satisfy the following properties. In order to list the properties of histories produced by executions of PSMVL, we need to include the operation f_i denoting the certification of T_i .

PSMVL₁: For every T_i , there is a unique starting timestamp $S_TS(T_i)$; that is, $S_TS(T_i) = S_TS(T_j)$ iff $i = j$.

PSMVL₂: For every T_i , f_i follows all of T_i 's reads and writes and precedes T_i 's commitment.

PSMVL₃: For every $r_i[x_j]$ in H , if $i \neq j$, then $c_j < r_i[x_j]$. That is, every read operation reads a committed version.

PSMVL₄: Let t_{now} be the time to execute $r_k[x_j]$. Then x_j is either (a) the most recently committed version before t_{now} or (b) the version that an active update transaction T_i whose security level is less than or equal to $L(T_k)$ reads down. In case (a), $C_TS(T_i) < C_TS(T_j)$ or $S_TS(T_k) < C_TS(T_i)$. In case (b), $C_TS(T_i) < C_TS(T_j) < S_TS(T_k) < C_TS(T_k)$ or $C_TS(T_j) < S_TS(T_k) < C_TS(T_i)$. That is, for every $r_k[x_j]$ and $w_i[x_i]$ in H , (a) $C_TS(T_i) < C_TS(T_j)$ or (b) $S_TS(T_k) < C_TS(T_i)$.

PSMVL₅: For every $r_k[x_j]$ and $w_i[x_i]$ (i, j , and k are distinct), either $f_i < r_k[x_j]$ or $r_k[x_j] < f_i$.

PSMVL₆: For every $r_k[x_j]$ and $w_i[x_i]$ in H , $i \neq j$ and $i \neq k$, if $r_k[x_j] < f_i$ and the priority of T_k is greater than that of T_i , then $C_TS(T_k) < TS(w_i[x_i])$.

PSMVL₇: For every update transaction T_i , there is a unique commit timestamp $C_TS(T_i)$. That is, $C_TS(T_i) = C_TS(T_j)$ iff $i = j$.

4 Examples

In this section, we illustrate the operations of the protocol by showing two example histories produced by PSMVL protocol. We show how each transaction reads the right version to meet various requirements.

$$T_1(P_1, L_1): r_1[x] c_1$$

$$T_2(P_2, L_2): r_2[x] w_2[y] c_2$$

$$T_3(P_3, L_3): r_3[x] w_3[x] c_3$$

	1	2	3	4	5	6
$T_1(P_1, L_1)$					$r_1[x_3]$	c_1
$T_2(P_2, L_2)$			$r_2[x_0]$	rejected		
$T_3(P_3, L_3)$	$r_3[x_0]$	$w_3[x_3]$		c_3		

Table 2: The first example

$$T_1(P_1, L_1): r_1[x] r_1[a] c_1$$

$$T_2(P_2, L_2): r_2[x] r_2[z] w_2[a] c_2$$

$$T_3(P_3, L_3): r_3[z] r_3[x] w_3[z] c_3$$

$$T_4(P_4, L_4): w_4[x] c_4$$

	1	2	3	4	5	6	7	8	9	10	11	12	13
$T_1(P_1, L_1)$							$r_1[x_0]$				$r_1[a_0]$		c_1
$T_2(P_2, L_2)$		$r_2[x_0]$							$r_2[z_0]$	$w_2[a_2]$		c_2	
$T_3(P_3, L_3)$	$r_3[z_0]$				$r_3[x_0]$	$w_3[z_3]$		c_3					
$T_4(P_4, L_4)$			$w_4[x_4]$	c_4									

Table 3: The second example

Example 1 Assume that $P_1 < P_2 < P_3$, $L_1 > L_2 > L_3$, $L(x) = L_3$, and $L(y) = L_2$. The operations of each transaction are specified as shown in Table 2.

In this example, $S.TS(T_1) = 5$, $C.TS(T_1) = 6$, $S.TS(T_2) = 3$, $S.TS(T_3) = 1$, and $C.TS(T_3) = 4$. Since $P(T_3) > P(T_2)$ and $L(T_3) < L(T_2)$, T_2 is rejected by T_3 at time 4 (by the rule in Figure 1 (a)).

Example 2 Assume that $P_1 > P_2 > P_3 > P_4$, $L_1 > L_2 > L_3 > L_4$, $L(z) = L_3$, $L(x) = L_4$, and $L(a) = L_2$. The operations of each transaction are specified as shown in Table 3.

At time 3, $HH\text{-list}(x) = \{T_2, T_3\}$. At time 4, since a HH/LL-conflict between T_2 and T_4 occurs, as shown in Figure 4 (a), $x_0 \rightarrow hhlptr$ points to a new node which contains L_3 and a count of 1. At time 5, T_3 reads x_0 because T_3 is in $HH\text{-list}(x)$. At time 6, $HH\text{-list}(z) = \{T_2\}$. At time 7, T_1 reads x_0 because $x_0 \rightarrow hhlptr$ is not null and it contains lower level transaction L_3 . At time 9, since $HH\text{-list}(z)$ is not null, T_2 reads z_0 and the versions of z are as shown in Figure 4 (b). At time 10, $HH\text{-list}(a) = \{T_1\}$ and at time 11, T_1 reads a_0 because T_1 is in $HH\text{-list}(a)$.

5 Correctness proofs

In this section, we prove that the PSMVL protocol guarantees one-copy serializability and no priority inversion. In addition, we show that it satisfies multilevel security requirements.

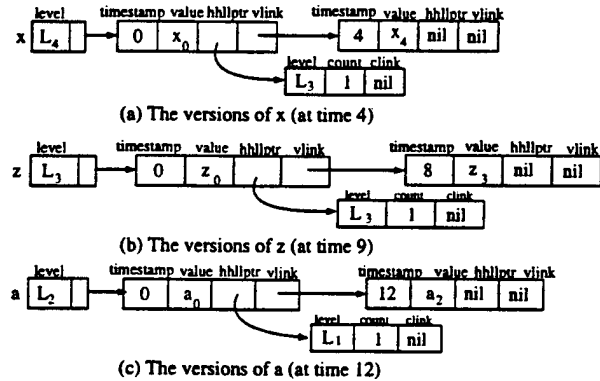


Figure 4: A sequence of operations for the second example

5.1 Serializability

Theorem 1 A multiversion schedule, H , is one-copy serializable (ISR) if and only if $MVSG(H, \ll)$ is acyclic [6].

Theorem 2 Every history produced by PSMVL is ISR.

Proof: Let T_1, T_2, \dots, T_n be a set of transactions, and H be a history produced by PSMVL protocol over T_1, T_2, \dots, T_n . We will prove that $MVSG(H, \ll)$ is acyclic by showing that every edge $T_i \rightarrow T_j$ in $MVSG(H, \ll)$ is in timestamp order.

We define a version order \ll by $x_i \ll x_j$ only if $C.TS(T_i) < C.TS(T_j)$. Suppose $T_i \rightarrow T_j$ is an edge of

$SG(H)^5$. This edge corresponds to a reads-from relationship (i.e., for some x , T_j reads x from T_i). Then, by $PSMVL_3$, $C_TS(T_i) < S_TS(T_j) < C_TS(T_j)$. Let $r_k[x_j]$ and $w_i[x_i]$ be in H where i, j, k are distinct, and consider the version order edge that they generate. There are two cases: (1) $x_i \ll x_j$, which implies $T_i \rightarrow T_j$ is in $MVSG(H, \ll)$; and (2) $x_j \ll x_i$, which implies $T_k \rightarrow T_i$ is in $MVSG(H, \ll)$.

Case (1) by definition of \ll , $C_TS(T_i) < C_TS(T_j)$.

Case (2) by $PSMVL_4$, either $C_TS(T_i) < C_TS(T_j)$ or $S_TS(T_k) < C_TS(T_i)$. The first case is impossible, because $x_j \ll x_i$ implies $C_TS(T_j) < C_TS(T_i)$. Hence, it must be true that $S_TS(T_k) < C_TS(T_i)$. We show that $S_TS(T_k) < C_TS(T_i)$ ensures $T_k \rightarrow T_i$ in $MVSG(H, \ll)$. There are two possible cases: (1) $P(T_k) > P(T_i)$ and (2) $P(T_k) < P(T_i)$.

In case (1), T_k starts before T_i commits, and $P(T_k) > P(T_i)$, and T_k reads the older version x_j rather than x_i . Therefore, it ensures that $T_k \rightarrow T_i$. In case (2), T_k has a lower priority and a higher security level than T_i . Thus, if T_i executes $w_i[x_i]$ before T_k commits, then T_k is aborted following the compatibility matrix in Figure 1 (a). However, there exists $r_k[x_j]$ in H . Thus, $C_TS(T_k) < S_TS(T_i) < C_TS(T_i)$ and it ensures that $T_k \rightarrow T_i$. Since all edges in $MVSG(H, \leq)$ are in timestamp order, $MVSG(H, \leq)$ is acyclic. By Theorem 1, H is 1SR. \square

5.2 Timing constraints

Theorem 3 A higher priority transaction is neither delayed nor aborted by low-priority transactions due to data contention on low-level data.

Proof: Let T_i and T_j be two transactions such that $P(T_i) > P(T_j)$ where $P(T_i)$ and $P(T_j)$ are the priorities of T_i and T_j respectively. Let $L(T_i)$ be the security level of T_i . There are three possible cases.

The first case is where $L(T_i) > L(T_j)$. When both T_i and T_j are about to access the same data item x , T_i reads down x while T_j writes into x because of their levels. Since $P(T_i) > P(T_j)$, by the version selection algorithm, T_i reads x written by T_k (not T_j) such that $C_TS(T_k) < S_TS(T_i)$. Thus, T_i is neither delayed nor aborted due to T_j . The second case is where $L(T_i) = L(T_j)$. Because T_i and T_j have the same level, they should be scheduled only by a protocol for RTDBMS that avoids priority inversion. Therefore, T_i is not aborted or delayed by T_j . The last case is where $L(T_i) < L(T_j)$. T_i has a higher priority than T_j . Hence, if T_i conflicts with T_j on the same data item x , T_j is aborted by T_i using the compatibility matrix in Figure 1 (a) and (d).

⁵ A serialization graph for a history H , $SG(H)$, is a directed graph whose nodes are transactions and whose edges represent all conflicting relationships between two transactions.

For all possible cases, high-priority transaction T_i precedes T_j . \square

5.3 Security properties

Theorem 4 No low-level transaction is ever delayed or aborted by a high-level transaction. In addition, a low-level transaction is not interfered with due to data contention by a high-level transaction.

Proof: By the MLS property, a transaction can read and write data items at its own level and only read down data items at lower levels. Let T_i and T_j be two transactions such that $L(T_i) > L(T_j)$ where $L(T_i)$ is the security level of T_i . If T_i and T_j are conflicting with each other, then we can see that T_i reads down the data item x while T_j writes into x . There are two possible cases.

The first case is when $P(T_i) < P(T_j)$. Because $L(T_i)$ is greater than $L(T_j)$ and $P(T_i)$ is less than $P(T_j)$, T_i is aborted or blocked according to the compatibility matrix in Figure 1 (a) and (d). Therefore, T_j is neither delayed nor aborted by T_i . The second case is when $P(T_i) > P(T_j)$. By the compatibility matrix in Figure 1 (b) and (c), T_j writes x without delaying and HH/LL-procedure is performed. Thus, T_j is neither delayed nor aborted by T_i . Since low-level transactions are neither delayed nor aborted, there is no security violations. \square

6 Performance evaluation

In this section, we present the simulation results to show the performance of PSMVL, compared with two other concurrency control protocols. The first protocol we compared with PSMVL is the Unconditional Multiversion Two Phase Locking (UMV2PL) protocol [10] for real-time databases. The UMV2PL protocol is based on MV2PL [6] and its compatibility matrix is shown in Figure 5. In UMV2PL, high priority transactions can abort low priority transactions in order to avoid priority inversion. However, when a high priority transaction requests a read or a certify lock, if a low priority transaction holds a certify lock, then the low priority transaction can convert its certify lock to a write lock for eliminating conflicts between two transactions. And thus, the UMV2PL reduces the number of the abortions of low priority transactions.

The other protocol is the 2PL-HP protocol [1] for real-time databases. The 2PL-HP protocol is based on the 2PL with a priority-based conflict resolution scheme to eliminate priority inversion. By comparing the performance of PSMVL with those protocols, the cost for satisfying security and timing requirements can be quantified.

6.1 Simulation model

In order to evaluate the performance of our protocol, we use SLAM II [3] and adopt the simulation model as shown in Figure 6 (a). The parameters used in the simulation study are presented in Figure 6 (b).

$T_R \backslash T_H$	R	W	C
R	Y	Y	Y*
W	A	Y	Y
C	-	Y	Y*

(a) Lock holder(T_H) has a lower priority

$T_R \backslash T_H$	R	W	C
R	Y	N	N
W	Y	Y	Y
C	N	N	N

(b) Lock requester(T_R) has a lower priority

Y : yes (allowed) N : No (not allowed) - : cannot occur
A : aborted (lock holder is aborted)
Y* : yes but a certify lock converts to a write lock

Figure 5: The compatibility matrix of UMV2PL

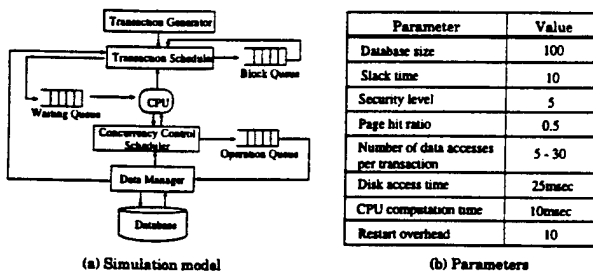


Figure 6: Simulation model and parameters

We compare PSMVL with UMV2PL and 2PL-HP in terms of the number of restart transactions, the average service time per transaction, and the fairness which shows how evenly the missed deadlines are spread across the input transactions of the various security levels. When a transaction is generated, it is delivered to a transaction scheduler which assigns a deadline and priority to the transaction as follows. Since we assume a soft deadline for each transaction, when a transaction misses its deadline, it is not aborted.

$$F_1. \text{DeadLine}(T) = \text{ArrivalTime}(T) + \text{SlackTime} * \text{TransactionSize}(T) * \text{CPUComputationTIME}$$

$$F_2. \text{Priority}(T) = \text{DeadLine}(T) * 100$$

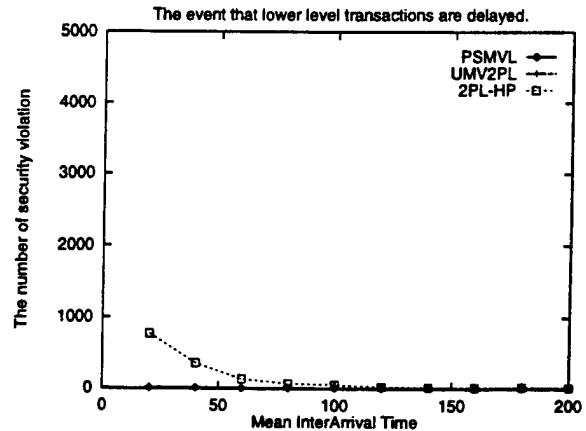
To compute the fairness, for each security level i , we use the formula,

$$F_3. \text{Fairness}(i) = \frac{\text{MissTrans}_i / \text{NoTrans}_i}{\text{MissTrans} / \text{NoTrans}}$$

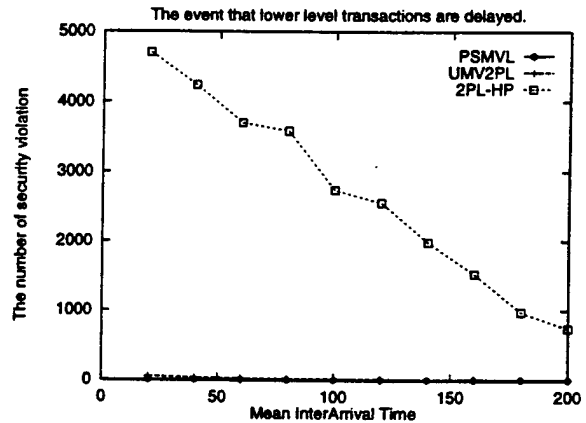
In the formula F_3 , MissTrans_i and NoTrans_i are the number of transactions at level i which miss the deadlines and the number of transactions at level i , respectively, whereas MissTrans and NoTrans are the total number of transactions which miss deadlines and the total number of all input transactions, respectively. If $\text{MissTrans} = 0$, then we let $\text{Fairness}(i)$ be 0.

6.2 Experimental results

The results of our performance analysis are shown in Figures 7, 8, 9, and 10.



(a) Transaction size = 10. Write operation ratio = 0.7.

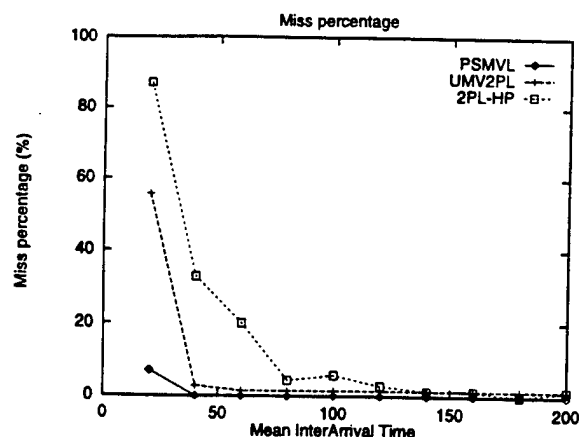


(b) Transaction size = 20. Write operation ratio = 0.7.

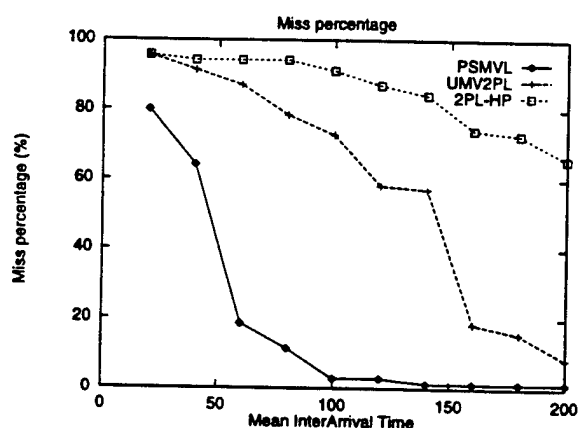
Figure 7: Security violation. Write operation ratio = 0.7

In Figure 7, we compare the three protocols PSMVL, UMV2PL, and 2PL-HP, in terms of the number of times that low level transactions are delayed or aborted by high level transactions. The x -axis represents the mean interarrival time (MIAT) which is the average time interval between the generations of transactions. If MIAT is small, then transactions are created more frequently. As shown in Figure 7, low level transactions are never delayed by high level transactions in PSMVL. On the other hand, UMV2PL, which is based on multiversion, has fewer blockings than 2PL-HP which is based on single version. The 2PL-HP has the worst performance because of its "wasted restarts". The transaction length in Figure 7 (a) is shorter than the length in Figure 7 (b). For short transactions, the number decreases rapidly when MIAT is in-

creased gradually, while the number decreases slowly for long transactions.



(a) Transaction size = 10. Write operation ratio = 0.7.



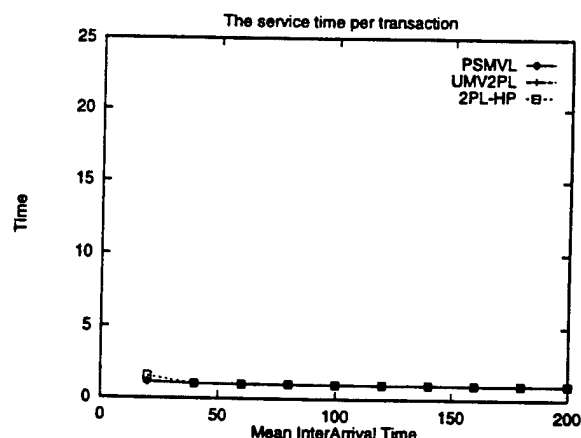
(b) Transaction size = 20. Write operation ratio = 0.7.

Figure 8: Miss percentage. Write operation ratio = 0.7

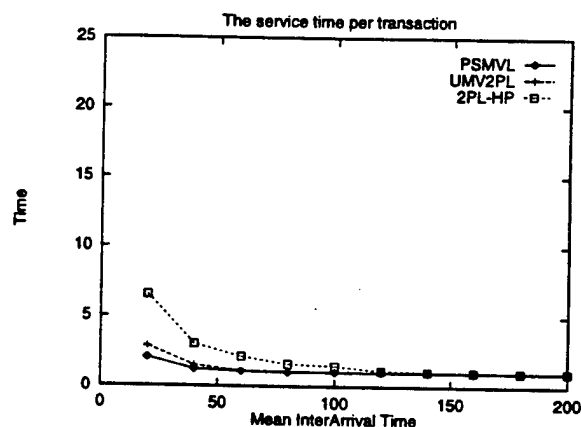
Figure 8 shows the percentage of transactions that miss their deadlines, denoted by *Miss Percentage*. Miss percentage is calculated with the following equation: $Miss\ Percentage = 100 * (\text{the number of tardy jobs} / \text{the total number of jobs})$. The number of $N_s(N_o)$ in the compatibility matrix of UMV2PL is more than that in the compatibility matrix of PSMVL. This causes UMV2PL to have more restart transactions than PSMVL. This is especially true for a high arrival rate, i.e., when MIAT is small, PSMVL shows better performance than UMV2PL. Therefore, a high arrival rate increases the number of restart transactions and results in high miss percentage.

When the transactions are short and the arrival rate of transactions is low, the miss percentage is rapidly reduced. However, for long transactions, the miss percentage is reduced more slowly. This indicates that long transactions is

one of the causes that increase the number of restart transactions. Since 2PL-HP uses a single-version, the number of restart transactions is higher when the transactions are scheduled using 2PL-HP, compared to UMV2PL.



(a) Transaction size = 10. Write operation ratio = 0.25.



(b) Transaction size = 15. Write operation ratio = 0.25.

Figure 9: Average response time. Write operation ratio = 0.25

Figure 9 shows the average service time per transaction. Let T_i be a transaction. Then, the average service time is $\frac{\sum_{i=1}^N (FinishTime(T_i) - StartTime(T_i))}{N}$, where N is the total number of transactions. The x and y axes represent interarrival times and average service times, respectively. Since we assume a soft deadline for each transaction, when a transaction misses its deadline, it is not aborted. Instead, it continues execution until its commitment. Therefore, transactions that miss their deadlines can be restarted several times. As shown in Figure 8, when the number of restart transactions increases, it takes more time to finish the transactions. PSMVL shows better per-

formance than UMV2PL, even though PSMVL has additional features such as security requirements. If the time interval between two transactions is short, the possibility of conflicts between transactions is increased.

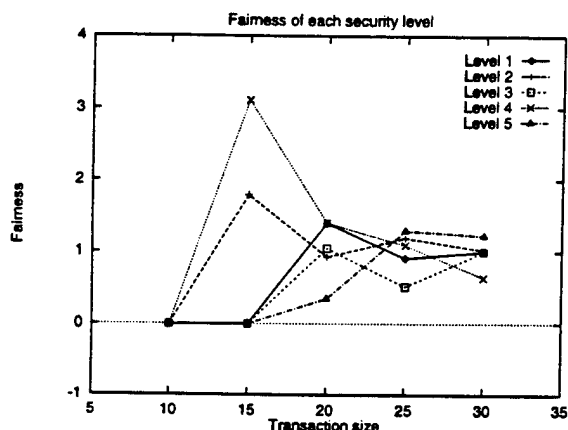


Figure 10: The fairness of the PSMVL. Mean InterArrival Time = 40

Figure 10 shows the fairness of the PSMVL. In the figure, the level 5 is the highest, while the level 1 is the lowest. When the transaction size is 15, only the transactions of level 2 and the transactions of level 4 miss their deadlines. This represents that in the PSMVL, the highest level transactions are not always sacrificed. And when the transaction size becomes smaller, the number of missed deadline transactions decrease. Therefore, if the number of deadline-missing transactions is small, then the divisor of the formula (F_3) is very small. As a result, for a security level i , Fairness(i) becomes big if the numerator of the formula (F_3) is not zero. Figure 10 also shows that when the transaction size increases, for each security level i , the value of Fairness(i) is getting closer. This means that the number of deadline-missing transactions is evenly distributed across the security levels and are not influenced by the security levels.

7 Conclusion

Database systems for real-time applications must satisfy timing constraints associated with transactions. Typically a timing constraint is expressed in the form of a deadline and is represented by a priority. In this paper, we have classified transaction processing systems according to their requirements and identified the conflicting nature of security requirements and real-time requirements. To address the problem, we have presented a new priority-driven multiversion locking protocol for scheduling transactions to meet their timing constraints in real-time secure database sys-

tems. The schedules produced by the protocol were proven to be one-copy serializable. We also presented our simulation model and evaluation results of the relative performance of the protocol, compared with other protocols.

The work described in this paper can be extended in several ways. First of all, in this paper we have not considered any trade-offs between real-time requirements and security requirements. A trade-off could have been made between those two conflicting requirements, depending on the specification of the application. For example, it would be interesting to see how a policy to screen out transactions that are about to miss their deadline would affect performance. Secondly, we have restricted ourselves by not distinguishing temporal and non-temporal data management. By exploiting the semantic information of transactions and the type of data they access, the protocol could be extended to provide a higher degree of concurrency. Finally, in this paper, we have restricted ourselves to the problem of real-time secure concurrency control in a database system. There are other issues that need to be considered in designing a comprehensive MLS/RT DBMSs, including architectural issues, recovery, and data models. We have started to look into those issues.

References

- [1] Abbott, R. K. and H. Garcia-Molina, "Scheduling Real-Time Transactions: Performance Evaluation", *Proceedings of the 14th VLDB Conference*, September 1988.
- [2] Abbott, R. K. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation", *ACM Trans. on Database Systems*, September 1992.
- [3] Alan, A. and B. Pritsker, *Introduction to Simulation and SLAM II*, Systems Publishing Corporation, 3rd edition, 1986.
- [4] Bell, D. E. and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations", *ESD-TR-73-278*, Mitre Corporation, 1973.
- [5] Bernstein, P. A. and N. Goodman, "Multiversion Concurrency Control - Theory and Algorithms", *ACM Transactions on Database Systems*, Vol. 8(No. 4), December 1983.
- [6] Bernstein, P. A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
- [7] Haritsa, J., M. Carey, and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems", *Real-Time Systems Journals*, Vol. 4(No. 3), 1996.

- [8] Keefe, T. F. and W. T. Tsai, "Multiversion Concurrency Control for Multilevel Secure Database Systems", *Proceedings of the 10th IEEE Symposium on Research in Security and Privacy*, May 1990.
- [9] Keefe, T. F., W. T. Tsai and J. Srivastava, "Multiversion Secure Database Concurrency Control", *Proceedings of the Sixth International Conference on Data Engineering*, Feb. 1990.
- [10] Kim, W. and J. Srivastava, "Enhancing Real-Time DBMS Performance with Multiversion Data and Priority Based Disk Scheduling", *Proceedings of the 12th IEEE Real-Time Systems Symposium*, December 1991.
- [11] Mukkamala, R. and Sang H. Son, "A Secure Concurrency Control Protocol for Real-Time Databases", *Proceedings of IFIP 9th Working Conference on Database Security*, August 1995.
- [12] Shu, L. C. and M. Young, "Correctness Criteria and Concurrency Control for Real-Time Systems : A Survey", *Technical Report No, SERC-TR-131-P*, November 1992.
- [13] Son, Sang H. and B. Thuraisingham, "Towards a Multilevel Secure Database Management System for Real-Time Applications", *Proceedings of IEEE Workshop on Real-Time Applications*, May 1993.
- [14] Son, Sang H., R. David, and B. Thuraisingham, "An Adaptive Policy for Improved Timeliness in Secure Database Systems", *Proceedings of IFIP 9th Working Conference on Database Security*, August 1995.
- [15] Son, Sang H., Seog Park, and Y. Lin, "An Integrated Real-Time Locking Protocol", *Proceedings of 8th International Conference on Data Engineering*, February 1992.

Multilevel Security
Chair: Pierangela Samarati

IRI: A Quantitative Approach to Inference Analysis in Relational Databases

Kan Zhang
Computer Lab, Cambridge University
kz200@cl.cam.ac.uk

Abstract

A new approach is introduced to evaluate inference risks in element-level labelling relational databases. Techniques from rough set theory are used to capture the *semantics of data*¹ and a quantitative measure Inference Risk Index (IRI) has been defined to characterise possible inference risks due to material implications reflected by the data. The approach is shown to be able to take into account of all *certain* and *possible* material implications in the data, including functional dependencies. It can also be used to address inference threats posed by rule-induction techniques from data mining. A major advantage of our approach is that the quantitative measure *IRI* is computed directly from data without knowledge input from System Security Officer. The computation is efficient and allows for real-time monitoring of inference risks during database run-time. Therefore, we are able to follow the changes in data patterns during database lifetime.

1 Introduction

In multilevel databases, inference has long been identified as a major threat to security. An inference problem in a multilevel database arises when a user with a low-level clearance, accessing information of low classification, is able to draw conclusions about information at higher classifications [16]. Marks [13] gives a formal definition of database inference, *Inference in a database is said to occur if, by retrieving a set of tuples $\{T\}$ having attributes $\{A\}$ from the database, it is possible to specify a set of tuples $\{T'\}$, having attributes $\{A'\}$, where $\{T'\} \not\subseteq \{T\}$ or $\{A'\} \neq \{A\}$* . In logic settings, we say there exists a *material implication*, denoted $(T, A) \Rightarrow (T', A')$, that relates the two sets. In other studies, material implications are sometimes referred to as *secondary paths* [1] or *inference channels* [16].

¹In this paper, we use the term *semantics of data* to mean the properties of data in KRS context, which is independent from the semantics of application represented by the relational database.

Here, we identify two types of inference problems. One is due to classification inconsistencies which arise from poor database design. In this case, *secondary paths* are formed by doing "joins" on the base tables in the databases to construct restricted tuples. Considerable research has been done to address this problem (see, for example, the work of Hinke [9], Thuraisingham [23], Binns [1], Burns [3], Garvey et al. [6], Hinke and Delugach [8], Lin [11], Qian et al. [16] and Rath et al. [17]). The aim of these works is to yield a well designed multilevel database in the sense that a user cannot, through any series of database queries, actually derive a restricted tuple.

The second type of inference problems is due to the semantics of application. Information stored in a relational database comprises of not only individual data items, but also the interrelationship among these data items. However, the relational data model emphasise efficient data structuring and manipulation. It lacks an adequate representation of data dependencies that characterise the application. Thus, multilevel classification of data items through element-level labelling is not enough to keep the data secret if the semantics of application or dependencies among data are not taken into account. The semantics of application is sometimes called *data dependencies*, *constraints*, or *rules* in different settings.

The second type of inference is harder to address since we need a way to find out and express the semantics of application in order to detect, analysis and eliminate the inference channels. In contrast, the approaches to address the first type of inference threats can be seen as syntactic in the sense that it can be done through schema manipulation without knowing the semantics of attributes.

Extensive work has been done to address the second type of inference. Among them, some approaches focus on eliminating the inference channels under the assumption that data dependencies are already known (see, for example, the work of Su and Ozsoyoglu [19], Stickel [22]). Others start from detecting and

analysing inference threats through various structures. The structures used to capture semantics of application include *Sphere Of Influence* (Morgenstern [14]), *Semantic Relationship Graph* (Hinke [9]), *Abductive Reasoning* (Garvey et al., [7]), *Semantic Net* (Thuraisingham [23]), *Graph* (Garvey et al. [6]), *Conceptual Graph* (Thuraisingham [23], Hinke and Delugach [8], Delugach and Hinke [5]), *Context* (Rath et al., [17]) and *Pattern* (Marks [13]). (Some of these techniques can also be used to address the first type of inferences.) However, available approaches have the following shortcomings.

- To some extent, all these approaches except [13] need help from System Security Officer (SSO) to generate the desired structures. The knowledge input from SSO represents the semantics of application. For example, in [6], the dependency of a flight's mission on its cargo has to be manually input into DISSECT in order for DISSECT to detect the inference channel between a flight's departure time and its mission. We think that if there is a relationship between a flight's mission and its cargo, this dependency must be reflected in the data in terms of material implication and we should be able to explicate it from the data directly. The fact that SSO can never be sure he knows all the dependencies among data means that available approaches only provide partial solutions. At most, they can claim that *to the best of our knowledge* there is no inference channel. There might well be data dependencies that the SSO is unaware of or are introduced into the database during its lifetime, which were not envisioned at database design time.
- Sometimes inference is *certain*, such as through functional dependency. However, more frequently we have cases in which inference is partial or with certain probability. Several previous works have addressed this situation, for example, Morgenstern [14], Garvey et al. [7] and Binns [1]. However, in existing approaches the probabilities are either assumed ([1]) or computed with the knowledge from SSO ([14], [7]). As in the previous point, we argue that if there is a probability associated with an inference path, this probability should be reflected by the data through material implication and can be explicated from the data itself.
- More recently, knowledge discovery in databases (KDD) or data mining (DM) techniques raise new security concerns for databases (Lin et al. [12]).

One of the primary approaches in KDD is *rule induction*, or learning from examples (see, for example, the works of Hu et al. [10], Srikant and Agrawal [18]). The derived generalized rules from KDD may open up new inference channels. Since the SSO may not be aware of such generalized rules, previous approaches to the second type of inference are inadequate in addressing threats posed by KDD or DM.

In this study, we propose an approach to detection and evaluation of the second type of inference threats that overcomes the above deficiencies.

2 Overview

We assume the *closed world assumption* (CWA). By CWA we mean that the data instances are complete and domain definitions are fully instantiated. Under CWA, all the *material implications* ([13]) corresponding to possible inferences can be derived from data. This does not mean all the knowledge needed to complete an inference chain has to reside in the database, i.e. the database doesn't have to contain all the semantics of an application. A chain of inference can be completed using outside knowledge. However, since the start and the end attribute values of an inference chain are in the database, there must be a material implication that corresponds to that inference chain and we should be able to discover that material implication from data under CWA. For detailed discussion of material implication and inference chain, please see [13].

Unlike previous approaches, here we are not trying to discover the semantics of application or the knowledge that can be used for possible logical inferences. As we have pointed out before, knowledge-based approaches may be incomplete since neither the database nor SSO may have/know all the semantics of application. Instead we use rough set theory as our tool to capture the semantics of data and quantify the inference risks through material implications. Since all logical inferences have corresponding material implications in the database, we are able to address all the possible inferences through material implications. Meanwhile, not all material implications have the corresponding logical inference paths, i.e. there may not be any apparent causal reason for a particular material implication. However, we think such material implications are still a legitimate concern for the current state of data in the database. We will say more about it in Section 5.

Rough set theory concerns the classificatory analysis of imprecise, uncertain or incomplete information.

It is a very effective methodology for data analysis and discovering rules in the attribute-value based domains. It is also an efficient tool for database mining in relational databases (Lin, [12]). This technique, which is complementary to statistical methods of inference, provides a new insight into properties of data. The main focus of this technique is on the investigation of structural relationships in data rather than probability distributions, as is the case in statistical theory. One of the main advantages of rough set theory is that it does not need any preliminary or additional information about data, such as probability distribution in statistics, or grade of membership or the value of possibility in fuzzy set theory.

The paper is organised as follows. In section 3, we give necessary concepts of rough set theory. In section 4 our approach is presented. Discussion of the approach follows in section 5. We draw our conclusions in section 6.

3 Basic Concepts of Rough Set Theory

Rough set theory was first introduced by Pawlak [15]. The primary problem addressed by the technique of rough sets is the discovery, representation and analysis of data regularities. The rough set-based methods are particularly useful for reasoning from qualitative or imprecise data.

3.1 Knowledge Representation System

In rough set theory, a *Knowledge Representation System*(KRS) is a quadruple

$$S = (U, A, V, f),$$

where U is a non-empty, finite set called universe, A is a finite set of attributes, $V = \cup V_a$ is a union of domains V_a of attributes a belonging to A , and $f : U \times A \rightarrow V$ is an information function such that $f(x, a) \in V_a$ for every $a \in A$ and $x \in U$.

The information function assigns attribute values to objects belonging to U . The Knowledge representation System allows for convenient tabular representation of data, which is similar to a relational table in the relational data base model (cf. Codd [4]). However, the relational model is not interested in the meaning of the information stored in the table. The emphasis is placed on efficient data structuring and manipulation. In the Knowledge Representation System the attribute values, i.e., the table entries, have associated explicit meaning as features or properties of the objects.

3.2 Indiscernibility Relation

With every subset of attributes P of A , we define an equivalence relation $IND(P)$, also called an *indiscernibility relation*, over U as follows:

$$(x, y) \in IND(P) \text{ iff } f(x, a) = f(y, a) \text{ for all } x, y \in U \text{ and } a \in P.$$

Equivalence relation $IND(P)$ induces a classification of objects into classes, each of which consists of objects with the same values of attributes belonging to P . Let U/P denote the family of equivalence classes of $IND(P)$. The equivalent classes of $IND(P)$ are also called *definable sets or concepts* of universe U using *knowledge* P . A KRS is selective iff all classes of U/A are one element set, i.e. $IND(A)$ is an identity relation.

The indiscernibility relation $IND(P)$ represents our ability to classify the objects in the universe using knowledge P . The central issue of rough set approach is the determination of how well a subset X belonging to universe U can be characterised in terms of the information available to represent objects of the universe U . With each subset $X \subseteq U$ and a subset of attributes $P \subseteq A$, we can associate two subsets:

$$PX = \bigcup \{Y \in U/P : Y \subseteq X\}$$

$$\bar{P}X = \bigcup \{Y \in U/P : Y \cap X \neq \emptyset\}$$

called the P -lower and P -upper approximation of X respectively.

Let x be in U and $P \subseteq A$ be our knowledge about universe U . We say that x is *certainly in* X using knowledge P iff $x \in PX$, and that x is *possibly in* X using knowledge P iff $x \in \bar{P}X$. Our terminology originates from the fact that we want to decide if x is in X on the basis of a definable set in S rather than on the basis of X . This means we deal with PX and $\bar{P}X$ instead of X . In the logic settings, it is equivalent to say that there are *certain* rules that classify x into X using knowledge P iff $x \in PX$, and that there are *possible* rules that classify x into X using knowledge P iff $x \in \bar{P}X$.

3.3 Dependency of Attributes

One of the main problems in the analysis of KRS with respect to discovering cause-effect relationships in data is the identification of dependencies among different groups of attributes.

Let C, D be two subsets of A and $C \cap D = \emptyset$, called *condition* and *decision* attributes, respectively. We introduce the notion of a positive region of U/D , $POS(C, D)$ as a union of lower approximations of all equivalence classes of the relation $IND(C)$:

$$POS(C, D) = \bigcup \{\underline{C}Y : Y \in U/D\}.$$

The positive region of U/D is a discernible part of U : that is, any object in $POS(C, D)$ can be uniquely

classified into one of the classes of U/D based solely on the knowledge of C , i.e. the values of attributes in C .

We say that the set of attributes D depends in degree k ($0 \leq k \leq 1$) on the set of attributes C in S if

$$k(C, D) = \text{card}(\text{POS}(C, D)) / \text{card}(U)$$

where card is the cardinality of a set. The value $k(C, D)$ provides a measure of dependency between C and D . If $k = 1$, then the dependency is full or functional; if $0 < k < 1$, then there is partial dependency; if $k = 0$, then the attributes C and D are independent. A dependency close to 1 gives reason to hypothesize that generally, there is a strong cause-effect relationship between attributes C and D , and a dependency close to 0 suggests weak, if any, cause-effect relationship between C and D .

3.4 Significance of Attributes

The relative contribution or significance of an individual attribute a belonging to C with respect to the dependency between C and D is represented by significance factor SGF , given by

$$SGF(a, C, D) = [k(C, D) - k(C - \{a\}, D)] / k(C, D)$$

if $k(C, D) > 0$.

Formally, the *significance factor* reflects the relative degree of decrease of dependency level between C and D as a result of the removal of the attribute a from C . In practice, the stronger the influence of the attribute a is on the relationship between C and D , the higher the value of the significance factor is.

4 A Quantitative Approach

In this work, we assume that data is stored in a relational database consisting of a series of tables and sensitive data items are protected by element-level labelling. Since we are not addressing the first type of inference problem identified above, we will further assume the universal relation paradigm [24] and view a relational database as a single table containing all the attributes from the entire database. We are not concerned about the actual mechanics of forming the view of a universal relation. For the second type of inference problem, we are only interested in evaluating the inference risks for those classified data items in the universal relation.

A relational table can be considered as a Knowledge Representation System in which columns are labelled by attributes, rows are labelled by the objects and the entry in column p and row x has the value $p(x)$. Each row in the relational table represents *information* about some object in universe U . However, the

relational model is not interested in the meaning of the information stored in the table. Consequently the objects about which information is contained in the table may not be represented in the table. Whereas in the KRS all objects are explicitly represented and the attribute values, i.e., the table entries, have associated explicit meaning as features or properties of the objects.

One way to conciliate the two models, which we adopt in this paper, is to take the primary key K in the relational table as object identifier and all other attributes ($A - K$) as attributes in KRS. In doing so, we may lose some information contained in the primary key attributes which is relevant to the semantics of the application. Another way to get around this is to add another attribute which assigns a unique identifier to each row of the relational table. In this case, all the attributes in the original relational table become attributes in the corresponding KRS and there is no information loss. However, in the relational data model there is always a primary key which identifies every object in the table, which means the derived KRS is selective. This situation may still happen even if we use primary key as object identifier. Selective KRS is not itself difficult to analyse, but we might have to lower the degree of precision in order to derive generalized rules. We will say more about it with the example later.

We should point out that in real applications, some data values of a relational table may be missing or imprecise. Consequently, the derived KRS is incomplete. For simplicity, in this paper we assume that all data items are precisely defined. Interested readers are referred to [20, 21] for dealing with uncertain data in rough set context.

4.1 Formal Specification

For simplicity, we assume a two-level labelling system, Classified and Unclassified. Given a universal relation R and its set of attributes A' , we view the relational table as a KRS in which every tuple of R is an object of universe U with the value x of primary key K as object identifier. The corresponding KRS has attribute set $A = A' - K$.

Instead of defining structures to detect and analyse inference risks, we choose to characterise the inference risk for each subset of classified attribute values belonging to the same object by an Inference Risk Index (IRI).

Definition For a classified data element set (x, P) in column set P and row x with value set P_x , let attribute set B denote the set of attributes whose values are classified for object x ($P \subseteq B$). Let attribute sets

$C = A - B$ and $D = P$. The associated Inference Risk Index $IRI(x, P)$ for data element set (x, P) is defined as

$$IRI(x, P) = \frac{card([x]_{IND(C)} \cap [x]_{IND(D)})}{card([x]_{IND(C)})}$$

where $card$ is the cardinality of a set, $[x]_{IND(C)}$ and $[x]_{IND(D)}$ are the equivalent classes of $IND(C)$ and $IND(D)$ containing x , respectively.

Intuitively, IRI gives us a quantitative measure of inference risks due to possible material implications existing in the database. It is computed from data directly.

From the definition of IRI , it is easy to see for any classified data element set (x, P) , $0 < IRI(x, P) \leq 1$. If $IRI(x, P) = 1$, we have $[x]_{IND(C)} \subseteq [x]_{IND(D)}$ which means there is a *certain* rule that can be induced from C to D for all the objects/tuples in $[x]_{IND(C)}$, i.e. there is a *certain* material implication $([x]_{IND(C)}, C) \Rightarrow ([x]_{IND(C)}, D)$.

Since it is always true that $x \in ([x]_{IND(C)} \cap [x]_{IND(D)})$, there is always a *possible* rule that can be induced from C to D for all the objects/tuples in $[x]_{IND(C)}$, i.e. there is a *possible* material implication $([x]_{IND(C)}, C) \Rightarrow ([x]_{IND(C)}, D)$. Therefore, $IRI(x, P)$ is always greater than 0. This is obvious since object/tuple x itself is a valid instance for the *possible* material implication.

Specifically, if $IND(C) \subseteq IND(D)$, i.e. D is functionally dependent on C , $IRI(x, P) = 1$ for any value of P . This conforms to our intuitive notion of functional dependency. Since attribute set C functionally determines attribute set D , given values of C the attacker should be able to infer values for D . Please note that there are cases where there is no functional dependency between C and D , but IRI still equal to 1. In these cases, there are *certain* rules between C and D that are valid only for the specific value of P_x , not for all the possible values of C and D in terms of the classical definition of functional dependency.

Finally, we should point out that our definition of IRI is conservative since $IND(C)$ and $IND(D)$ are calculated with full knowledge of relevant data in the database. In reality, an attack can only see part of the data needed.

4.2 Algorithm

We present an outline of the process of computing IRI . Optimal algorithms are our current research topic.

Step 1: Decide attribute sets C and D .

Step 2: Decide the most significant attribute set C_s of C .

Step 3: Compute IRI according to definition using C_s instead of C .

Step 4: (optional) Generalize some of the attributes in C or D and repeat the whole process.

If we skip Step 2, we are able to find all the material implications existing in the data. However, in real applications, not all these material implications accurately capture the dependencies of data. Some irrelevant attributes may contribute, even though in a negligible way under CWA, to the discernibility of knowledge C , therefore, disturb the real dependency we are trying to express using IRI . The purpose of Step 2 is to remove these noises that may disguise true dependencies in the data. This is more useful for smaller databases.

Sometimes, the discernibility due to attributes of the derived KRS is very high. We may find a large amount of material implications, but each of them may just have a small number of valid instances in the database. This might not be interesting since we might want to know the inference risks due to qualitative rules. In this case, we can try to generalise some of the attributes in order to discover and evaluate qualitative rules.

It is apparent that the above algorithm permits fast and efficient evaluation of IRI . In fact, one of the advantages of rough set theory is that programs implementing its methods can easily run on parallel computers. Therefore it is possible to monitor inference risks during database run-time by real-time evaluation of IRI . In doing so, we are also able to follow the changes in data patterns during database lifetime.

4.3 An Example

EMP_ID	Grade	Location	Salary
1	2	CA	8000
2	3	NY	6000
3	5	TX	5000
4	8	TX	2000
5	2	CA	7000(C)
6	8	CA	2000
7	5	TX	5000(C)
8	5	CA	4000
9	7	NY	3000
10	7	MA	3000

Figure 1: Original Employee Table

Consider the multilevel relational table shown in Figure 1. The relation can be seen as a KRS with primary key *EMP_ID* as object identifier. The attributes of the KRS are *Grade*, *Location* and *Salary*. Now we want to calculate *IRI* for object 5 and classified value 7000 of attribute *Salary*. First we use $C = \{Grade, Location\}$ and $D = \{Salary\}$ to evaluate *IRI*. We can derive directly from the table that $U/C = \{\{1, 5\}, \{3, 7\}, \{2\}, \{4\}, \{6\}, \{8\}, \{9\}, \{10\}\}$ and $U/D = \{\{1\}, \{2\}, \{3, 7\}, \{4, 6\}, \{5\}, \{8\}, \{9, 10\}\}$. Therefore, we have $IRI(5, \{Salary\}) = 0.5$. Similarly, we can find $IRI(7, \{Salary\}) = 1$.

If we take a closer look at the significance of attributes *Grade* and *Location*, we can find that $SGF(Grade, C, D) = 0.88$ and $SGF(Location, C, D) = 0.38$. Therefore we may remove attribute *Location* and use $C_s = \{Grade\}$ to evaluate *IRI*. Since $U/C_s = \{\{1, 5\}, \{2\}, \{3, 7, 8\}, \{4, 6\}, \{9, 10\}\}$, we have $IRI(5, \{Salary\}) = 0.5$ and $IRI(7, \{Salary\}) = 0.67$. These figures give us a more accurate evaluation of the dependency between *Grade* and *Salary*.

EMP_ID	Grade	Location	Salary
1	2	CA	HIGH
2	3	NY	HIGH
3	5	TX	MIDDLE
4	8	TX	LOW
5	2	CA	HIGH(C)
6	8	CA	LOW
7	5	TX	MIDDLE (C)
8	5	CA	MIDDLE
9	7	NY	LOW
10	7	MA	LOW

Figure 2: Generalized Employee Table

We can also generalise some of the concepts in the data to evaluate inference risks due to qualitative rules. Suppose salary range is defined to be *HIGH* = 5500+, *MIDDLE* = 3500 – 5500, and *LOW* = 1500 – 3500. (Generalisation can also be done with attributes in *C*.) Thus the original table in Figure 1 turns into one shown in Figure 2. It is immediate that attribute *Grade* functionally determines attribute *Salary*. Not surprisingly, we find $IRI(5, \{Salary\}) = 1$ and $IRI(7, \{Salary\}) = 1$, which means that there are generalized *certain* rules

that can be induced from attribute *Grade* to *Salary*. These generalized rules are more likely to be true since they are based on larger amount of valid instances.

5 Discussion

As we have explained before, the second type of inference threats arise not from penetration of the security mechanisms directly but rather from the very nature of the semantics of application. In classical terms, inference in relational databases is used to refer to logical process of proving or deriving some classified attribute values for some tuples from some unclassified attribute values. In this sense, inference is associated with some type of causal relationship, such as functional dependencies. As has been pointed out by Marks in [13], material implication is more general than functional dependency. Material implications only require that sets of data and attributes occur together, regardless of whether one causes the other, both are caused by a third activity, or they occur by coincidence.

In this study, we take a different view of inference threats from that of Marks [13]. We think that if a material implication is valid for the data in a database, it should be considered as a possible inference path and should be taken into account by SSO. There might not be logical reasons for a particular material implication. However, the lack of causal reasons may due to the limits of SSO's understanding of the semantics of application. For example, in the case of scientific data, causality is what scientists are trying to discover in the research process. On the other hand, even if a material implication is just a coincidence, the very fact that this coincident is valid in the current state of data means that it is a legitimate concern. The material implication may be picked up by an attacker's data mining tools and used to derive a valid classified association unexpectedly. The attacker may not necessarily believe the results, but, at least, the validity of the material implication alarms the attacker that this could be true.

We notice that in [13] Marks insightfully formalises general inference as *material implication*. He goes on using *Patterns* to detect *certain* material implications reflected in the data, which correspond to the cases where $IRI = 1$ in our approach. The rough set based approach proposed in this paper is able to capture both *certain* and *possible* material implications reflected in the data.

In addition, our approach is able to address inference risks due to generalized rules by lowering the representation accuracy. The rough set approach that underlies the whole technique is based on the intuitive

observation that lowering the degree of precision in the representation of objects, for example, by replacing the numeric temperature measurements by qualitative ranges of HIGH, NORMAL, or LOW, makes the data regularities more visible and easier to characterise in terms of rules. Lowering the representation accuracy, however, might lead to the undesired loss of information expressed in the reduced ability to discern among different concepts. To analyse and evaluate the effect of different representation accuracies on concept discernibility levels, a number of analytic tools have been developed [25]. By using these tools, one can attempt to find a representation method that would compromise between sufficient concept discernibility and the ability to reveal essential data regularities. It is, in fact, the ability of rough set based approach to generalise the concepts that enables it to deter attempts to use generalized rules discovered by data mining for inference [10, 18].

In computing *IRI*, deciding the most significant subset of *C* is a tricky part. We are investigating optimal algorithms. In this paper we considered functional dependency in relational databases through material implication. Since various database dependencies can be represented using indiscernibility relations [2], we plan to consider other dependencies, such as multivalued dependencies, in the future.

6 Conclusion

In this paper, we proposed a new approach to evaluation of inference risks in element-level labelling relational databases. Techniques from rough set theory are used to capture the semantics of data and a quantitative measure Inference Risk Index (*IRI*) has been defined to characterise possible inference risks due to material implications reflected by the data. The approach is shown to be able to take into account of all *certain* and *possible* material implications in the data, including functional dependencies. It can also be used to address inference threats posed by rule-induction techniques from data mining.

A major advantage of our approach is that the quantitative measure *IRI* is computed directly from data without knowledge input from System Security Officer. The computation is efficient and allows for real-time monitoring of inference risks during database run-time. Therefore, we are able to follow the changes in data patterns during database lifetime.

Acknowledgments

I thank Roger Needham for helpful discussions and Ken Moody who clarified some of my misconceptions.

References

- [1] L. Binns, Inference through secondary path analysis, *Proc. Sixth IFIP Working Conf. Database Security*, Vancouver, B.C., Canada, Aug. 1992.
- [2] W. Buszkowski and E. Orlowska, On the Logic of Database Dependencies, *Bulletin of Polish Academy of Sciences, Mathematics*, Vol. 34, No. 5-6, 1986.
- [3] R.K. Burns, A conceptual model for multilevel database design, *Proc. Fifth Rome Laboratory Database Security Workshop*, Fredonia, NY, Oct., 1992.
- [4] E.F. Codd, A Relational Model of Data for Large Shared Data Banks, *Comm. ACM*, Vol. 13, pp. 377-387, 1970.
- [5] H.S. Delugach and T.H. Hinke, Wizard: A Database Inference Analysis and Detection System, *IEEE Trans. on Knowledge and Data Engineering*, Vol. 8, No. 1, Feb 1996.
- [6] T.D. Garvey, T.F. Lunt, X. Qian, and M. Stickel, Toward a tool to detect and eliminate inference problems in the design of multilevel databases, *Proc. Sixth IFIP Working Conf. Database Security*, Vancouver, B.C., Canada, Aug. 1992.
- [7] T.D. Garvey, T.F. Lunt and M.E. Stickel, Abductive and Approximate Reasoning Models for Characterising Inference Channels, *Proc. of the Computer Security Foundations Workshop IV*, 1991.
- [8] T.H. Hinke and H.S. Delugach, AERIE: An Inference Modelling and Detection Approach for Databases, *Proc. Sixth IFIP Working Conf. Database Security*, Vancouver, B.C., Canada, Aug. 1992.
- [9] T.H. Hinke, Inference Aggregation Detection in Database Management Systems, *Proc. 1988 IEEE Symposium on Security and Privacy*, April 1988.
- [10] X. Hu, N. Shan, N. Cercone and W. Ziarko, DBROUGH: A Rough Set Based Knowledge Discovery System, *Proc. 8th Int'l Symp. on Methodologies for Intelligent Systems*, Charlotte, NC., USA, 1994. (LNCS 869)
- [11] T.Y. Lin, Inference Secure Multilevel Databases, *Proc. Sixth IFIP Working Conf. Database Security*, Vancouver, B.C., Canada, Aug. 1992.

- [12] T.Y. Lin, T.H. Hinke, D.G. Marks, and B. Thuraisingham, Security and Data Mining, *Proc. Ninth IFIP Working Conf. Database Security*, Aug. 1995.
- [13] D.G. Marks, Inference in MLS Database Systems, *IEEE Trans. on Knowledge and Data Engineering*, Vol. 8, No. 1, Feb 1996.
- [14] M. Morgenstern, Controlling Logical Inference in Multilevel Database Systems, *Proc. 1988 IEEE Symposium on Security and Privacy*, 1988.
- [15] Z. Pawlak, Rough Sets, In *Theoretical Aspects of Reasoning About Data*. Kluwer, Netherlands, 1991.
- [16] X. Qian, M.E. Stickel, P.D. Karp, T.F. Lunt, T.D. Garvey, Detection and Elimination of Inference Channels in Multilevel Relational Database Systems, *Proc. 1993 IEEE Symposium on Security and Privacy*, 1993.
- [17] S. Rath, D. Jones, J. Hale and S. Sheno, A Tool for Inference Detection and Knowledge Discovery in Databases, *Proc. Ninth IFIP Working Conf. Database Security*, Aug. 1995.
- [18] R. Srikant and R. Agrawal, Mining Generalized Association Rules, *Proc. of the 21st Int'l Conference on Very Large Databases*, 1995.
- [19] T-A. Su and G. Ozsoyoglu, Data Dependencies and Inference Control in Multilevel Relational Database Systems, *Proc. 1987 IEEE Symposium on Security and Privacy*, 1987.
- [20] R. Slowinski, J. Stefanowski: Rough classification in incomplete information systems, *Mathematical and Computer Modelling* 12 (1989) no.10/11, 1347-1357.
- [21] R. Slowinski, J. Stefanowski: Rough-Set Reasoning about Uncertain Data, *Fundamenta Informaticae*, 27(2/3): 229-243 (1996)
- [22] M.E. Stickel, Elimination of Inference Channels by Optimal Upgrading, *Proc. 1994 IEEE Symposium on Security and Privacy*, 1994.
- [23] B. Thuraisingham, The Use of Conceptual Structures for Handling the Inference Problem, *Proc. fifth IFIP Working Conf. Database Security*, Shepherdstown, WV, November 1992.
- [24] J.D. Ullman, *Principles of Database and Knowledge-Base Systems*, vols. I and II, Rockville, MD.: Computer Science Press, 1988, 1989.
- [25] W. Ziarko, The Discovery, Analysis, and Representation of Data Dependencies in Databases, in *Knowledge Discovery in Databases*, G. Piatetsky-Shapiro and W.J. Frawley, (eds) Menlo Park, CA: AAAI/MIT, 1991, 195-209.

Role-based Access Control

Chair: Elisa Bertino

Software Architectures for Consistency and Assurance of User Role-Based Security Policies

S. A. Demurjian, Sr., T. C. Ting, and J. A. Reisner
Computer Science & Engrg. Dept.
The University of Connecticut
Storrs, Connecticut 06269-3155
{steve,ting,reisner}@eng2.uconn.edu

Abstract

How will assurance and consistency be attained during the definition and usage of an application's user-role based security policy, particularly in an object-oriented context that stresses change and evolution? An important question, especially with an exploding interest in designing/developing object-oriented software in C++, Ada95, and Java. Security concerned users and organizations must be provided with the means to protect and control access to object-oriented software. Our approach to user-role based security (URBS) for object-oriented systems and applications has emphasized:

- a customizable public interface that appears differently at different times for specific users, to control and limit access;
- security policy specification via a user-role hierarchy to organize and assign privileges (public interface methods) based on responsibilities; and,
- extensible/reusable URBS enforcement mechanisms that utilize inheritance, generics, and exception handling for the automatic generation of code for the URBS security policy.

This paper expands our previous work to include assurance and consistency, particularly since we are committed to a continued exploration of automatically generating URBS enforcement mechanisms. This paper employs the field of software architectures to explore intriguing solutions that further our URBS/object-oriented efforts.

1 Introduction

How will assurance and consistency be attained during the definition and usage of an application's user-role based security policy, particularly in an object-oriented context that stresses change and evolution?

This question is interesting, particularly with the explosive growth of object-oriented software development. While C++ has been a strong player since the late 1980s, Ada95 and Java offer new opportunities that are targeted for diverse and significant market segments. Ada95 and its strong ties to DoD and government software, and Java with an increasing impact on commercial internet-based and general-purpose software, both expand the base of software professionals working on object-oriented platforms. Security has been a paramount concern, especially in Java, where security must be present to control the effects of platform-independent software. Organizations will demand high consistency and high assurance in object-oriented software, across a wide range of domains. Health care systems

require both high levels of consistency and assurance, while simultaneously needing instant access to data in life-critical situations. In CAD applications, the most up-to-date specifications on mechanical parts must be available in a shared manner to promote cooperation and facilitate productivity, making consistency and assurance important from a business perspective.

Unfortunately, security has often been an afterthought in the design and development process. For example, in databases, it is often after relations have been implemented and instances populated into the database, that the security issues are considered by database programmers writing transactions, rather than by the security engineer, where the responsibility should truly rest. It is our strong belief that a cohesive and comprehensive security policy can be an important means of identifying the set of allowable users and insuring that there is a solid definition of each user's capabilities. Security must be an equal partner during design and development, allowing the impact of the policy on all application components to be understood.

Over the past few years, we have concentrated on discretionary access control, by defining a user-role based security (URBS) model that can be utilized in the design and development of object-oriented systems and applications. The current public interface provided by most object-oriented languages is the union of all privileges (methods) needed by all users of each class. This allows methods intended for only specific users to be available to all users, i.e., there is no way to prevent access by any user to a method in the public interface. For example, in a health care application (HCA), a method placed in the public interface to allow a Physician (via a GUI tool) to prescribe medication on a patient can't be explicitly hidden from a Nurse using the same GUI tool. Rather, the software engineer is responsible for insuring that such access does not occur, since the object-oriented programming language cannot inherently enforce the required security access. Our approach promotes a customizable public interface that appears differently at different times for specific users. We have proposed a user-role definition hierarchy (URDH) to organize responsibilities and to establish privileges. Privileges can be assigned (can invoke a set of application methods) or prohibited (cannot invoke a set of application methods) to roles. Our recent efforts have proposed extensible and reusable URBS enforcement mechanisms that utilize inheritance, generics, and exception handling for the automatic generation of code from the URDH. Our goal has been to minimize the amount of knowledge a software engineer must have on URBS by having mechanisms that are self-contained, class libraries, which supply all of the required code to define and enforce the desired security policy. Work on the object-oriented design model [5, 11] and URBS [1, 9, 10] have been published.

This paper expands our previous work to include assurance and consistency, particularly since we are committed to a continued exploration of automatically generating URBS enforcement mechanisms. We believe that class libraries may not offer a secure enough venue to insure high consistency and assurance for enforcement mechanisms. Thus, we have turned to the field of software architectures to investigate potential solutions to augment our previous URBS enforcement approaches [2, 3, 4]. Software architectures [16] expand traditional software engineering by looking at how different major system components can mesh and interact. This is especially relevant for object-oriented software, where a class library for a problem is initially developed, with software engineers designing and building tools against that class library to implement the overall capabilities of an application. In such a model, the URBS enforcement mechanism must interact with both the class library and the tools for an application, to insure that users utilizing tools only access those portions of the application on which they have been granted access. In our approach, this translates to the users only being able to invoke methods that have been authorized to their respective roles.

The remainder of this paper contains five sections. In Section 2, we provide background on the ADAM environment [5]. In Section 3, we discuss the critical need of consistency for security, as we seek to guarantee a level of assurance to designers and users utilizing an URBS/object-oriented approach. In Section 4, we briefly review two of our previous URBS enforcement approaches, propose and explore software architectural variants that can offer varying degrees of assurance and consistency, and critique the variants by comparing and contrasting their capabilities from multiple perspectives. Section 5 examines related work in URBS for object-oriented systems, looking at their support for consistency and assurance. Section 6 concludes this paper.

2 Background: The ADAM Environment

ADAM [5, 11] is an integrated system for language-independent, object-oriented design environment for applications that have software engineering, database, and security requirements. ADAM, short for Active Design and Analyses Modeling, can generate compilable code in C++ (GNU C++ and Ontos C++ - an object-oriented database system), Ada83, Ada95, and Eiffel for any object-oriented design. An integral part of this process is the definition of URBS [9, 10]. Both Unix and PC versions of ADAM are available. As an example, a health care application (HCA) is employed [9], as shown in Figure 1.

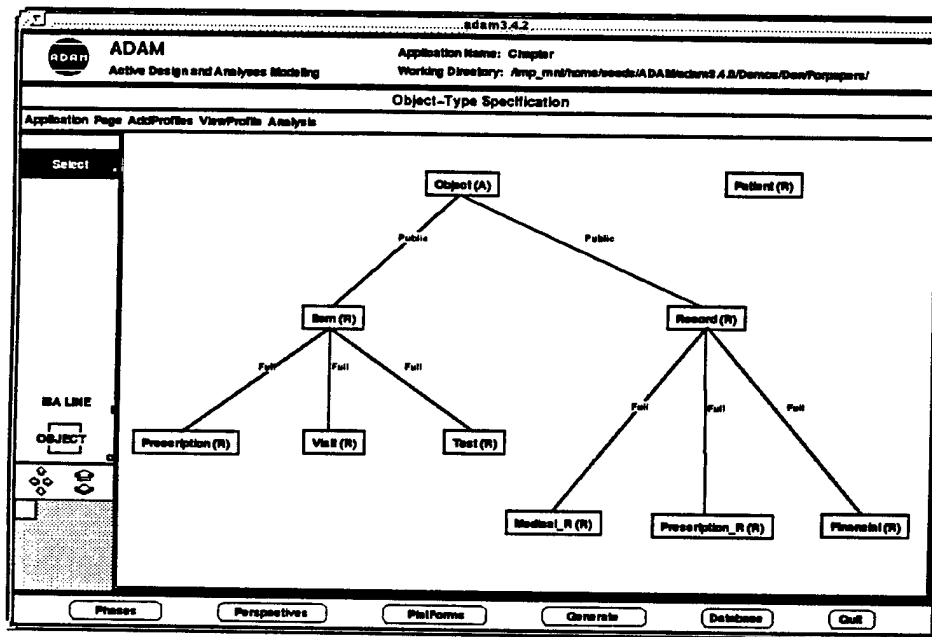


Figure 1: Sample Object Types for HCA.

Figure 2: Sample Object, Attribute, and Method Profiles for HCA.

To track the purpose and intent of different design choices and constructs in an application, *profiles* are utilized [5, 9]. Profiles force software engineers to supply detailed information as an application is designed, and are used to facilitate on-demand and automatic feedback via analyses to alert designers

whenever an action in the environment results in a conflict or possible inconsistency. The object-type profile for Prescription, the attribute profile for Cost, and the method profile for UpdateCost are shown in Figure 2. Other profiles have been omitted for brevity.

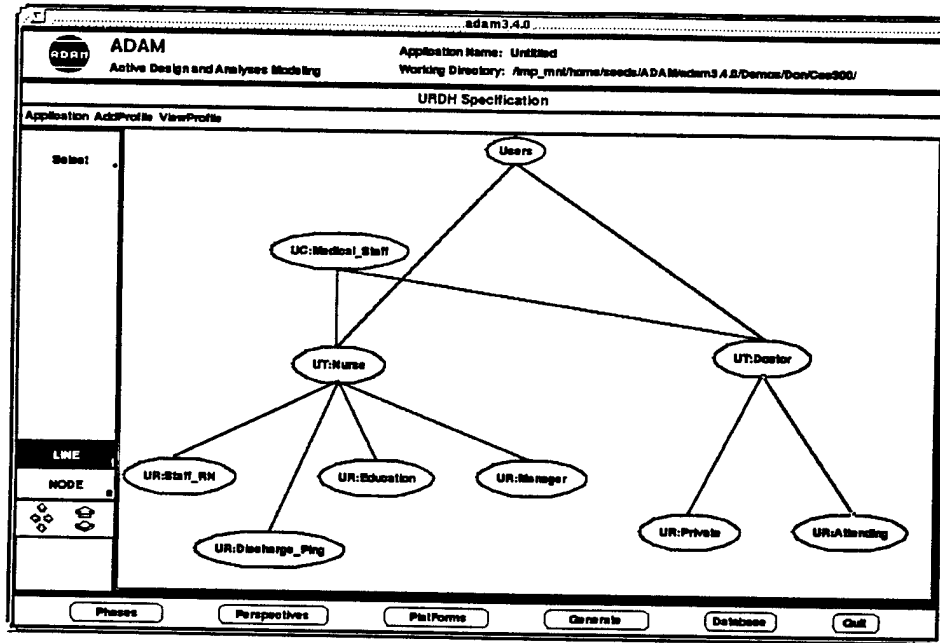


Figure 3: The URDH of the HCA.

To support URBS, the user-role definition hierarchy (URDH) characterizes the different kinds of individuals (and groups) who all require different levels of access to an application. Figure 3 shows a partial URDH in ADAM for HCA. *User roles (UR)* (e.g., *Staff_RN*, *Education*, etc.), can be grouped under a single *user type (UT)* (e.g., *Nurse*). When multiple UTs share privileges, a *user class (UC)* can be defined (e.g., *Medical_Staff*). To define, UCs, UTs, and URs, we utilize a *node profile (NP)*: 1. a name for the node; 2. a prose description of its responsibility; 3. a set of assigned methods (the positive privileges); 4. a set of prohibited methods (the negative privileges); and 5. a set of consistency criteria for relating URDH nodes.

A node description for a UT in Figure 3 is: *Nurse: Direct involvement with patient care on a daily basis*. In addition, for each UR, the role-security requirements are defined, e.g., *Staff_RN: All clinical information for the patients that they are responsible for. Can write/modify portions of clinical information to track patient progress. Cannot change a Physician's orders on a patient*. To establish privileges, application methods are assigned to URDH nodes. Commonalities (shared methods) can be moved from a set of URs to their shared UT, and from a set of UTs to their shared UC. Methods shared by all UTs can be moved up to Users. Thus, commonalities flow up the URDH, while differences flow down. Prohibited methods are used to explicitly identify which methods cannot be accessed by a URDH node. *Equivalence (subsumption) criteria* allow the security engineer to identify which URs (UTs/UCs) must have the same (subsumable) capabilities, as reflected in the assigned/prohibited methods. Conflicts between assigned and prohibited methods or violations of consistency criteria are flagged by ADAM.

The final step in ADAM involves user authorization, which is accomplished via a *user profile (UP)*: 1. a name for the user; 2. a prose description of its responsibility; 3. a prose description of its security requirements; 4. a set of assigned URs (the positive privileges); 5. a set of prohibited roles (the negative privileges); and 6. a set of criteria for relating users. Note that conceptually, a user has privileges via a set of one or more assigned URs. So, the URDH information is aggregated for each UR for which a user has been authorized (or prohibited).

3 The Need for Assurance, Consistency, and Analyses

Role-based security policies and enforcement mechanisms must have high consistency in order to support a high assurance, secure system. The consistency must be maintained at all levels within the policy, including individual roles, role hierarchies, and end-user authorizations, to insure that their creation, modification, and deletion will always maintain the required URBS policy. Consistency is the foundation upon which high integrity and assured secure systems must be built. A set of techniques/tools must be provided that allow URBS policies to be analyzed and assured at all times during design, development, and maintenance of object-oriented software.

In general, URBS policies are application dependent, and consequently, data security requirements vary widely from application to application. For example, sensitive health care data must be both protected from unauthorized use while simultaneously be almost instantaneously available in emergency and life critical situations. On the other hand, in some design environments such as CAD, the most up-to-date specifications on mechanical parts must be available in a shared manner to promote cooperation and facilitate productivity. In this case, the URBS policies may not protect sensitive personal information, but may protect information which is equally sensitive from a business perspective. Further, the strength of URBS policies is that they are intended to be specific to individuals, determined based on individual needs and special conditions. It is this idiosyncrasy that makes it so difficult to address the issues of designing consistent URBS policies.

The ultimate responsibility for URBS policies is on the shoulders of the application's management personnel and organization's data security officer. In order to have these critical policy makers take full advantage of URBS, tools and techniques must be made available. Design techniques, similar to the ones presented in Section 2, are critical to allow software and security engineers to accurately and precisely specify their applications' functional and security requirements. To augment these techniques, a suite of tools is required, that can provide many different and diverse analytical capabilities. These tools should automatically alert these engineers when potential conflicts occur during the creation or modification of roles, role hierarchies, and end-user authorizations, thereby heading off possible inconsistencies. There must also be tools that provide on-demand analyses, allowing engineers to gauge their realized software and/or security requirements against their specifications. Once the URBS policy has stabilized, the tools should provide the means to capture and realize it via a URBS enforcement mechanism that is automatically generated. The overriding intent is to finish with an object-oriented system that embodies a strong confidence with respect to the URBS policy and its attainment.

The remainder of this section explores these and other issues from two perspectives. From the user-role definition perspective, in Section 3.1, we examine the consistency issues that must be attainable as roles and dependencies among roles are created and modified. From an authorization perspective, in Section 3.2, we investigate similar consistency issues as actual individuals (people) are authorized to play certain roles within an object-oriented application or system.

3.1 Consistency for User Roles

When a security engineer is creating and modifying user roles for an object-oriented application or system, the consistency of the definition is critical in order to insure that the URBS policy is maintained. This is a time-oriented issue; changes to the policy are needed, especially in object-oriented situations, where evolution and extensibility are the norm. Regardless of the changes that are made, there must be assurance that the privileges of each user role are adequate to satisfy the functions of the user role. Moreover, the privileges must not exceed the required capabilities of the user role, to insure that misuse and corruption do not occur. In addition, since user roles are often interdependent upon one another (e.g., our approach uses a hierarchy), it may be necessary to examine their interactions to insure that privileges aren't being passed inadvertently from role to role, yielding a potentially inconsistent state.

There are many different scenarios of evolution that must be handled. A security engineer may create new roles for a group of potential users or may create specific roles that are targeted for a particular end-user for a special assignment under a special circumstance. Each newly created role must be *internally consistent* so that no conflicts occur within the role itself. This is also true when a role is modified, which we term *intra-role consistency*. For the object-oriented case, when privileges

are assigned to each role, this assignment implicitly grants object-access privileges to the role holder (end-user). Such an assignment process utilizes the *least-privilege principle* which grants only necessary access privileges but no more. Only those privileges that are relevant to the user role are permitted. The policy is intentionally very conservative and restrictive, requiring that the URBS policy be validated by either the software engineer, security engineer, or both. In some organizations, there are dedicated security officers who possess the ultimate responsibility with respect to security requirements/policies for all applications.

To complement the least-privilege principle, user roles often must satisfy *mutual exclusion conditions*. Here, there must be a careful balance between permitting access to certain objects while simultaneously prohibiting access to other, special objects. Mutual exclusion is a strong URBS concern, and may be dictated by an organization's rules and regulations, or by government law. For example, in HCA, an individual assigned the role of Pharmacist can read the prescription of a patient, update the number of refills after processing the prescription, but is explicitly prohibited from modifying the dosage or drug of the prescription. Thus, access and modification to some information is balanced against exclusion from other information. This strong mutual exclusion situation is clearly observed by the medical profession and is mandated by law. The URBS policy must ensure that security requirements such as these are not violated. In our approach, these mutual exclusions are supported in the URDH by allowing the security engineer to define prohibited methods. Such a technique is extremely important to insure that the objects that are referenced within the prohibited privileges of a role do not overlap or contradict with the objects that are accessible by all of the assigned privileges.

When one extrapolates to consider the interdependence of user roles, such as within our user-role definition hierarchy (URDH), the internal consistency as captured by least privilege and mutual exclusion, must be expanded to *inter-role consistency*. In any approach with interdependence among user roles, there is the potential for user roles to acquire privileges (both positive and negative privileges) from other roles. This acquisition process must be clearly understood by the security engineer, particularly in an environment where URBS policy is constantly changing. In addition, to provide versatile design tools to the security engineer, it should be possible to establish superior, inferior, and equivalence relationships among different user roles. These relationships must also be validated as privileges are defined, acquired, and change. From the perspective of the entire URDH, *intra-hierarchy consistency* must be attained.

To support a URBS definition process with least privilege and mutual exclusion, the security engineer must be provided with a set of techniques and tools. From a purely definitional perspective, there must be tools for meaningful comprehension on user roles, including all positive privileges, negative privileges, and relationships to other roles. Such a role-profiling tool provides a quick and comprehensive display on the capabilities of each user role, supporting intra-role consistency. Once any initial definition has occurred, there must be tools to support analyses for both internal and inter-role consistency. Even if one has given minimal access to certain objects to a specific user role, it is the nature of object-oriented applications to be coupled. Hence, even a minimal set of explicit object accesses might correspond to implied access to a wider set of objects. Automated analysis tools are necessary for an exhaustive search to follow all possible object access paths as required by all of the positive and negative privileges in the security definition. Conflicts discovered during the search will have to be resolved by the application's management personnel and security engineer. Feedback must be available to assist the human designer in arriving at a viable resolution to any conflicts or inconsistencies.

Analyses are available in the ADAM environment for the application's content/context, and for its security requirements [5, 9], and is supported via the profiles reviewed in Section 2. Capabilities analyses in ADAM allows the security engineer to review the permissions (as inferred by the assigned and prohibited methods) given to a chosen URDH node on an application's OTs, methods, and/or private data, thereby supporting the intra-role or internal consistency of the URBS policy. Authorization analyses in ADAM allows the security engineer to investigate which user roles have what kinds of access to different aspects of an application (i.e., an OT, a method, or a private data item). Through these analyses, inter-role and intra-hierarchy consistency can be understood, since the security engineer can examine the multiple roles that access a component of the application. Overall, the analyses are intended to provide a crucial first step in the important process of assurance with respect to the

consistency and correctness of the URBS policy.

3.2 Consistency in End-User Authorization

When considering consistency in end-user authorization, the assumptions of the policy must be clearly understood. For example, in any organization where end-users can be assigned multiple roles, there are two scenarios of permissible behavior against an application: 1. End-users can only play exactly one role at any given time; and, 2. End-users can play multiple roles concurrently at any given time. The first assumption does not cause significant problems, since for an end-user, only one role is active. As long as that role is intra-role and inter-role consistent, there is no problem. However, the first assumption alone does not provide the needed security, but instead raises a number of interesting issues that are addressed by the second assumption.

Namely, when an end-user that may play multiple user roles simultaneously at any given time within an application and within the organization, a level of *end-user consistency* is introduced. Similar in concept to inter-role consistency, in end-user consistency, the privileges of the multiple roles for a single end-user are aggregated. Such an aggregation may introduce conflicts between positive and negative privileges that span multiple roles. Further, when a new privilege is assigned to an established user role, with internal and inter-role consistency assured, it may still impact the end-user consistency. Also, when a new role is assigned to an end-user, it too may conflict with existing concurrent roles for that user. While all of these different problems are the responsibility of the security engineer, given the dynamics and complexity of a real-world organization, it would be very difficult for that person to accomplish this task without appropriate and effective tools.

Automated tools are needed for the user authorization model in a secure data system so that no URBS policy violations are possible for any end-user in the organization. Thus, the techniques/tools in Section 3.1 must be extended to consider end-user consistency, allowing the security engineer to focus on the conflicts of privileges for single end-users with multiple concurrent roles. Tools from both definitional and analytical perspectives are required. ADAM provides such tools as extensions of the URDH case described in Section 3.1, since they (in most cases) repeatedly call the "relevant" URDH analyses for each user role assigned to an individual.

Once intra-role, inter-role, and intra-hierarchy, and end-user consistency have been attained at a definitional level, there are two remaining requirements:

1. the defined URBS policy must be captured within the object-oriented application; and
2. once captured, at both compile time and runtime, the policy must be enforced.

For both requirements, our previous work on URBS enforcement approaches [2, 3, 4] is intended to support, in part, the consistency and assurance of the URBS policy. However, as we will see in Section 4, through software architectures we can provide a higher level of assurance regarding the guarantee that must be met concerning a defined URBS policy for an object-oriented application. Our approach will involve the encapsulation of the URBS policy and enforcement mechanism into a software component that is part of a larger software architecture for the support of an object-oriented application with embedded security. It is our hope that such a software component can be effectively managed, controlled, and most importantly validated.

4 Software Architectures and URBS Mechanisms

To understand our efforts in this section, it is critical that we define our assumptions concerning the composition of object-oriented software. Basically, the crux of an object-oriented system is an underlying, shared object type/class library to represent the kernel or core functionality, e.g., in HCA, the OTs given in Figure 1. Once such a library has been developed, other software engineers will design and develop tools against it, e.g., in HCA, a patient user interface, an admissions subsystem, a software component to allow a blood analysis system to send test results directly into patient records, etc. Thus, in our approach, end-users are not able to write programs to access data directly, which is instead the

responsibility of software engineers. Rather, end-users utilize tools that embody the apropos security code to enforce the required URBS policy.

Software architectures [16] is an emerging discipline whose intent is to force software engineers to step back from their traditional algorithm/data structure perspective and view software as a collection of interacting components. Interactions occur both locally (within each component) and globally (between components). In understanding interactions, the key consideration is to identify the communication and synchronization requirements which will allow the functionality of the system to be precisely captured. By taking a broader view of the problem definition process, software architectures permits database needs, performance/scaling issues, and security requirements, to be considered. These considerations are critical as large-scale object-oriented software becomes more dominant in industry.

Our purpose in this section is to present and critique multiple software architectural variants for URBS enforcement of object-oriented software, with a constant focus on the attainment of consistency and assurance. While our previous efforts focused on detailed URBS enforcement approaches [2, 4], our intent in this section is to step back from this work and consider the ways that these approaches can fit into an overall architectural scheme to security enforcement for object-oriented software. Nevertheless, we start this section by briefly reviewing two of our previous approaches, since they set the context for our subsequent discussion related to software architectures. Then, we focus on two architectural styles: layered systems, which are most known from ISO layers; and, communicating processes, which underlie today's client/server paradigm. For both styles, multiple variants are presented and analyzed. The final section critiques the six different variants by comparing and contrasting their capabilities with respect to: the ability to support consistency/assurance, the impact of evolving both the security policy and application classes, the usability of our two previous approaches to enforcement, and, the presence of a database.

4.1 UCLA and GEA Approaches

Two approaches to enforcement were previously presented [2, 4], and are briefly summarized in this section. The *URDH-class-library approach (UCLA)* employs inheritance to implement the enforcement mechanism by creating a class hierarchy for the URDH. For each URDH node, positive method access is based on the defined assigned methods. As the application executes, method invocations must validate against the current UR by checking the method call against the URDH class library. At runtime, a user's role guides the invocation of the appropriate methods that are used to verify whether the user's role has the required permissions. From an evolvability perspective, as URs are added, only the URDH class library must be recompiled. Similarly, if a UR's assigned/prohibited methods change, only the URDH classes would need to be modified and recreated. Changes to the application still impact on both class libraries.

The *generic exception approach (GEA)* incorporates concepts of reusable template classes to realize a significant core of generic code that encapsulate the URBS policy. In GEA, when a method is invoked, the UR of the current user is checked to verify if access can be granted. If not, an exception is raised and processed. When the GEA security template is utilized by a class that needs protection, it is customized based on type, resulting in compile and runtime changes to the code. This customization insures that a check method is called when a user attempts to invoke any method on the active instance. If the user's role doesn't permit access, an exception is thrown, and the invoking method will not allow its functionality to be executed and affect instances. There are many advantages to GEA. First, the code in the GEA security template is hidden from the software engineer. Software reuse is promoted since the template is reused by all classes that require URBS enforcement. Second, once the UR has been established, the remaining code works based on that initialization. While the code for OT/class methods must be changed, the changes are hidden from the software engineer writing tools.

4.2 Architectural Alternatives

From a software architecture perspective, the URBS enforcement mechanism can be located in many different places and function in many different ways. It can be integral part of the OT/class library, to

be automatically included when any tool utilizes a portion of the library. Alternatively, it may be an independent and self-contained library that is compiled with each application tool, similar in concept to a math library being included. Other choices could have a separately executing process through which all security requests must be handled. Regardless of the choice, the key underlying characteristics must be the attainment of high consistency and assurance. This must be balanced against concerns that seek to minimize the amount of knowledge a software engineer must have on URBS and to yield an approach that is evolvable, since object-oriented software (and its security policy) must be conducive to change.

To standardize terminology regarding the assumptions on object-oriented systems given in the introduction of Section 4, we define:

AppCL: Represents the shared, object-oriented class library for an application, e.g., the HCA classes as shown in Figure 1.

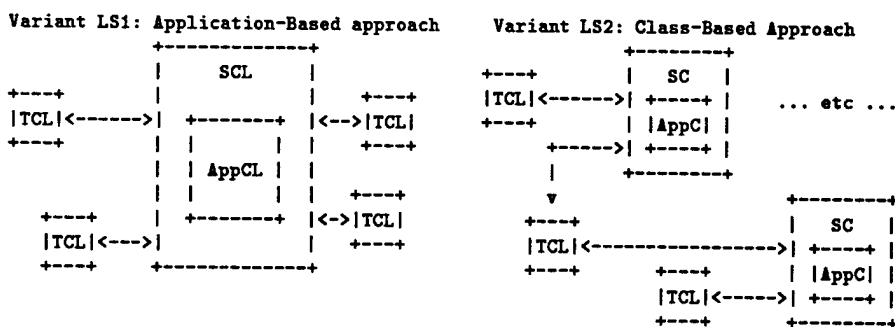
SCL: Represents the security class library for an application that embodies URBS definition and enforcement, e.g., the classes generated from the URBS definition as given in Figure 3.

TCL: Represents the tool class library for individual tools (e.g., a patient GUI, an admissions subsystem, etc.) against the application.

Note that when the L is dropped from either AppCL or SCL, we are referring to an individual class of the library. Using these basic assumptions, the remainder of this section explores layered systems, and communicating processes and the client/server paradigm, as software architectural alternatives for URBS enforcement. For each alternative, multiple variants are presented and discussed, and then analyzed with respect to: the level of consistency and assurance that each variant provides for security concerned users; the dimensions of evolvability, which is critical since both the URBS policy and object-oriented software tend to be dynamic over time; and, the impact of the absence/presence of a persistent store (i.e., database).

4.2.1 Layered Systems

Layered systems are a classic technique for software architectures, where layers of functionality are built upon one another to provide a controlled environment for access to information. There are two layered system variants for URBS enforcement: LS1 an application-based approach and LS2 a class-based approach. In both variants, security is at the level of the method invocation, which is processed by the SCL prior to its actual runtime call against an instance of a class in the AppCL. In either case, the SCL can be either the UCLA or GEA approach. In the LS2 variant, each individual class handles the method invocations that apply to its instances as they are received by the various tools. The difference is one of granularity. In LS1, security is managed at the application level overall, and once it has been determined that the tool can invoke a method, it is passed through to the involved instance or instances. In LS2, security is managed at the instance level only. This may cause a problem when instances refer to other instances, i.e., a security request by the tool involves multiple instances of either the same or different classes.



From a consistency/assurance perspective, it appears that LS1 has the advantage, since all of the method invocations must pass forward through the security layer for authentication and all results must pass back for enforcement. That is, when utilizing a tool and its various options, users end up calling various methods based on his/her UR and under the control of the tool. However, variant LS2's view of allowing each instance to maintain its own security is superior to LS1 from a software evolution perspective, since changes to the security policy of one class may not effect the policies of other classes. When a persistent store is included into the mix, LS1 has the edge, since all accesses must proceed via a common security layer. In LS2, there are potential concurrent access issues if some or all AppCs are directly connected to a database.

4.2.2 Communicating Processes - C/S Paradigm

In the *communicating processes* approach to URBS enforcement, a process-oriented, client/server (C/S) paradigm is adopted. TCL, SCL, and AppCL are integrated into single and/or multiple processes, resulting in a total of four different variants:

Variant CP1: Base case with a single process that combines TCL, SCL, and AppCL.

Variant CP2: Multi-process with TCL and SCL combined into a client that is served by AppCL.

Variant CP3: Multi-process with TCL a client of a combined SCL and AppCL server.

Variant CP4: TCL, SCL, and AppCL are independent processes that interact in a two level C/S architecture.

Note that for both CP1 and CP2, each SCL and/or AppCL represents the minimal subset needed by the tool/TCL to support its functionality and enforce its security policy.

Variant CP1 is similar to LS1, but each tool is compiled as a separate, standalone process. In this case, SCL and AppCL are analogous to a math library that is compiled when needed by the software. Functionally, within each process, TCL sends method invocations to SCL which in turn passes them through to AppCL according to the URBS policy. Results are passed back from AppCL to SCL, which may then filter the response before passing them back to TCL. Note that SCL and AppCL in each process represents those subsets of the class libraries needed by each tool, and that either UCLA or BEA can be the enforcement mechanism realized within SCL.

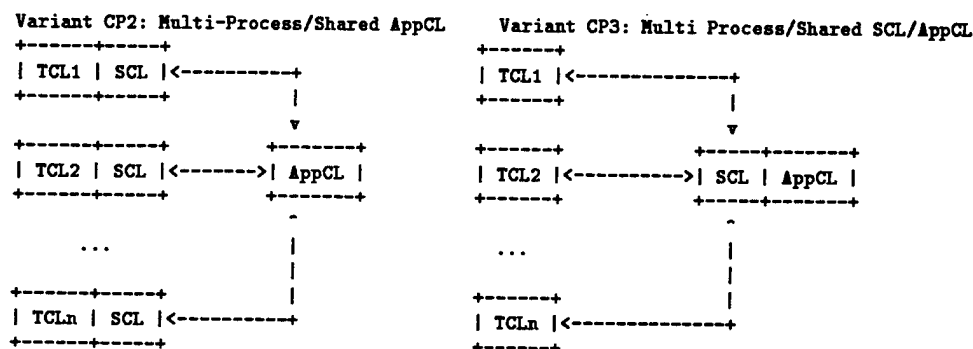
Variant CP1: Single Process - Base Case - no C/S

-----+-----	-----+-----		-----+-----
TCL1 SCL AppCL	TCL2 SCL AppCL	...	TCLn SCL AppCL
-----+-----	-----+-----		-----+-----

From a consistency/assurance perspective, it would be a requirement that each tool be compiled into a single process with SCL and AppCL included. Thus, the level of assurance and consistency that is attained is tied to the accuracy and completeness of the URBS policy. But, note that since each tool may have a only a portion of the overall URBS policy, consistency becomes a prominent concern whenever changes need to be made, i.e., updates must be made to all tools that use the portion of the policy that changed. Extensibility in CP1 presents major problems. While it is easy to add new tools, and new tools when added won't effect existing tools, changes to either the SCL or AppCL definitely cause problems. If changes to the SCL are localizable to data files that can be dynamically loaded, then URBS policy changes should be supportable. But, if the changes require the SCL to be rebuilt, unless the compilation/runtime environment supports dynamically linkable class libraries, evolution will also require the recompilation of all affected tools. More significantly, changes to AppCL have a dramatic impact for all affected tools and all affected portions of the SCL. In addition, since the AppCL is compiled with each tool, it is unclear whether this approach can successful work when AppCL is linked to a database.

Variants CP2 and CP3 are both multi-process approaches with clearly defined client/server separation of functionality. In CP2, each client is a TCL/SCL pair that interacts with a shared AppCL server. In this case, each SCL represents that subset of the overall URBS policy/enforcement that is needed by the

specific tool, i.e., if a tool only uses one or two classes, the SCL is that subset of the overall URBS policy for those needed classes. Thus, the URBS policy/enforcement is specifically bound to each tool. Like CP1, the level of consistency/assurance that is attained depends on the realization of the URBS policy within SCL. The fact that the policy is spread across multiple tools does introduce potential consistency concerns when changes to the policy are made. Changes to the URBS policy impact SCL in the same way as CP1. However, there are improvements in changes to AppCL; since it is in a separate process, careful planning will allow some changes to have no impact on the joint TCL/SCL clients. Drastic changes to AppCL (e.g., deletion of classes, additions of classes, major functionality upgrades) are likely to impact SCL thereby requiring the recompilation of tools. As conceptualized, both UCLA and GEA are tightly linked to AppCL, making them inappropriate for CP2. From a database perspective, the presence of a persistent store within or coupled to AppCL should be supportable and invisible to the clients.



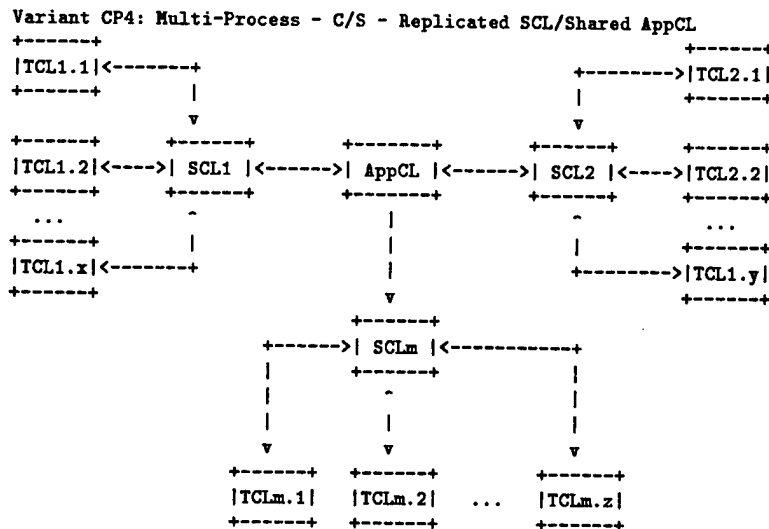
In CP3, the client is each individual tool (TCL), with the server containing the joint SCL/AppCL functionality. By decoupling the URBS policy/enforcement from each tool, the tool becomes relatively independent from changes to the security policy. Each tool simply makes requests to the joint server and the way that those requests are satisfied can be hidden using typical object-oriented design approaches. Thus, unlike CP1 and CP2, changes to the URBS policy shouldn't impact tool code. The placement of the entire URBS policy/enforcement in one location greatly improves consistency and assurance, since all changes to the policy occur in one place. This is superior to both the CP1 and CP2 variants. Like CP1, SCL can be realized with UCLA or GEA.

Changes to the URBS policy and/or the AppCL may require that the joint server be periodically rebuilt, i.e., changes to AppCL may still impact SCL. As long as those changes don't alter the signatures of the various methods/protocols that that tools utilize, there should be no impact on the tool code. Basically, the dimension of evolvability allows the easy addition of new tools or new users utilizing existing tools. Database integration of AppCL is the same as CP2. However, from a performance perspective, since all security requests are processed by a joint server, there is the potential that that server will become a bottleneck as the throughput of the system increases, i.e., with more tools, or more users utilizing existing tools.

Variant CP4 is presented as a means to alleviate the remaining consistency, assurance, and performance concerns of CP3. Variant CP4 is truly a multi-process, multi-leveled, client/server architecture. In this case, each TCL is a client to an SCL server that provides security for the entire AppCL, i.e., the SCLi's are replicated. Each SCL, in turn, is a client to the shared AppCL. Like CP2, SCLi's separation from AppCL negates UCLA and GEA as appropriate solutions.

The relationship between each TCLi.j and its respective SCLi acquires the advantages of CP3 with respect to: the independence of the tool code from SCL (and AppCL); the ability to add new tools; and, the lack of impact of changes to SCL (and AppCL) on the tool code. The multiple SCL servers to the TCL clients also alleviate a level of performance concerns from CP3, allowing more SCLs to be added as more tools (and hence, more users) need to be served. Consistency and assurance in CP4 maintain the benefits of CP3 over the other two variants: each SCL has the entire URBS policy/enforcement, so any changes to the policy can be made and replicated. CP4 still may have performance bottlenecks with

respect to access to AppCL. But those bottlenecks have now been delineated from the SCLs, and can be handled by replacing AppCL by a distributed object-oriented class library with database support.



4.3 Critiquing the Architectural Variants

This section summarizes the evaluative statements for the six variants into a cohesive discussion that clearly compares and contrasts their capabilities. Our first critique is based on the location and structure of the URBS policy/enforcement within each variant. This is important from a consistency and assurance perspective. In LS1, CP3, and CP4, the entire policy/enforcement is present and captured within SCL (replicated in CP4). In LS2, CP1, and CP2, the policy is partially captured, to the level required by the tool/TCL. From a consistency perspective, whenever the URBS policy changes, there must be assurance that the policy is still enforced by all existing tools. The centralized nature of LS1, CP3, and CP4, lends itself to a maintenance of the assurance after the change. In the case of LS2, CP1, and CP2, the tools/TCLs must be recompiled to insure that all SCLs are updated. Also, since the policy is spread across multiple SC/AppC pairs (in LS2) or is unique to each process (in CP1 and CP2), there is a chance that inconsistencies can arise that impact on assurance, if all recompilations are not carefully performed.

Our second critique involves the impact of changes on each variant when either the security policy or application classes are changed. For LS1, LS2, CP3, and CP4, as long as accepted object-oriented design techniques (abstraction, representation independence, etc.) have been followed, it should only be necessary to recompile SCLs and/or AppCLs; there should be no impact on tools/TCL. In fact, depending on the actual enforcement approach (UCLA, GEA, or other), two situations might occur: when the security policy changes, SCL or SCL/AppCL may need recompilation; and, when some application classes change, AppCL or AppCL/SCL may need recompilation. Both situations are dependent on the interrelation of the enforcement approach to the application classes. For other variants: when the policy changes, CP1 and CP2 must be rebuilt, since SCL is within the same process/client as the tool/TCL; when some application classes change, each tool/TCL in CP1 that uses the subset that has changed must be recompiled. CP2 behaves in a similar fashion to CP3 and CP4 for changes to the AppCL.

A third critique involves the utility of our existing enforcement mechanism approaches (UCLA and GEA) for the architectural variants. As currently designed, both UCLA and GEA are tightly coupled to AppCL. That is, it would be difficult to cleanly and completely separate out the SCL from the AppCL. This being the case, it is apparent that some variants are more conducive to the two approaches than others. Namely, LS1, LS2, CP1, and CP3, can all function with either UCLA or GEA as SCL, since SCL is linked to AppCL. On the other hand, neither CP2 nor CP4 can support UCLA and GEA for the AppCL, without changes to UCLA and GEA that decisively separate the security policy/enforcement

from the application class library. It will be necessary to either rework UCLA and GEA, or design new enforcement variants to support CP2 and CP4.

Our final critique focuses on the case when database interactions are required from the AppCL to a persistent store. LS1, CP2, CP3, and CP4 all separate AppCL from the tools/TCL, meaning that a persistent store can be easily supported. LS2 and CP1 have problems, since each approach utilizes a partial AppCL, for only those classes that are needed by each tool/TCL. Thus, for LS2 and CP1, if database access was to occur, it would likely require that the tools interact to synchronize their requests, which raises many major roadblocks. From a performance perspective, of the variants that can easily support a database, all but CP4 have potential bottlenecks at either the SCL, AppCL, or both. CP4 offers the best solution, and if needed, the AppCL can be expanded to a distributed object-oriented database to satisfy increases in either tools or users.

5 Pertinent Research Works in This Area

One recent effort in security for distribute objects raises the issue of high assurance as a key requirement [7]. Interestingly, their approach utilizes a layered architecture as a means to realize, in part, security for interacting, heterogeneous objects. The layered architecture contains: an underlying theoretical model (layer 1), a meta-object model (layer 2) as a primitive object architecture that is used by an abstract object model (layer 3) to construct distributed object-oriented applications that execute under the control of a object-oriented OS environment (layer 4). High assurance in their approach is predicated on the innermost theoretical layer that provides the formal methods that both capture and enforce the security policy. While their approach is object-oriented, it is not clear whether mandatory access control and/or URBS will supported.

There has also been a significant body of research related to our own efforts in role-based security for object-oriented systems. While none of these efforts specifically address consistency and assurance, there are inferences that can be drawn regarding their potential to support these two important concepts. In [13, 14], a model of authorization for next-generation database systems is proposed. In this model, an authorization is defined as a 3-tuple (s, o, a) , where s belongs to the set of subjects, o belongs to the set of authorization objects in a system, and a belongs to the set of authorization types. The roles are defined to reduce the number of authorization subjects and these roles form a role lattice, which is similar to the URDH in our approach. The authorization can be either implicit/explicit, strong/weak, or positive/negative, the latter of which is similar to assigned/prohibited methods. Collectively, their different combinations of an authorization are strongly related to consistency/assurance, since they imply a certain, specific behavior of their enforcement process. For example, implicit authorization is utilized to infer additional authorization requirements from a stored set of base privileges. In their approach, the allowable combinations of the different authorizations are critical for a security engineer to successfully define his/her policy.

In object-oriented software, the presence of inheritance introduces a potential set of problems that can significantly impact on the security [17]. For example, a user may have access authorized on the subclass B, but have access denied on the superclass A. The possible problems include a potential information flow from a higher security level to a lower level by inheriting the data item and a potential write-down of information by using the methods defined in a higher level to update the data item in the lower level. The identification and resolution of inheritance problems are critical to promote consistency and facilitate assurance of the URBS policy/enforcement mechanism. Assigned/prohibited methods, the URDH, and analyses can be used by a security engineer to avoid inheritance related problems.

Another effort has examined the issues and considerations that are required to integrate a collection of existing, locally defined user views into a single shared view by utilizing an object-oriented approach [6]. When different views are merged, there is the requirement to identify problems and inconsistencies, in order to resolve any conflicts that might exist among the different views. The merging of views to resolve conflicts is a key step towards a consistent and assured URBS policy. In this sense, their work is similar to our analysis techniques. However, their work is geared towards retrofitting a shared schema on top of existing, different views, in a traditional database design framework, while ours is

based on a shared schema and the allocation of privileges from this common basis for general purpose object-oriented applications.

Other related work similar to our own efforts has been in the object-oriented programming area. There has been work on *aspects* as a mechanism for object-oriented data models to extend a given class with new capabilities, including, roles [15]. Another effort allows different classes in an application to have different subjective views [8]. An extension of this effort has focused on composing these subjective views with implementation support in C++ [12]. None of these efforts approach the problem from either user role-based security and/or database perspectives. Thus, it is unclear what level of consistency and assurance can be supported.

6 Concluding Remarks and Future Work

Consistency and assurance for object-oriented systems is critical, since it is their nature to evolve and change over time. When both the application class library and the URBS policy are dynamic, those changes have the potential to significantly impact on the application's tools, which in turn, impacts on actual users. The emerging discipline of software architectures can be utilized to examine alternative placements for the tools, URBS policy, and application class library. In Sections 4.2 and 4.3, we presented and critiqued six different software architectural variants, based on layered systems (2 variants) and communicating processes (4 variants). Based on our analyses it appears that three approaches rank comparably: LS1 - a layered system with a shared, URBS policy/enforcement and application class library that is utilized by multiple application tools; CP3 a client/server solution where each tool is a client to a server that consists of a joint process containing the URBS policy/enforcement and application class library; CP4 a multi-level, client server solution where each tool is a client, the URBS policy/enforcement is replicated as a server, and the application class library has its own independent server. The only real difference between the three occurs when a database interacts with the application class library, causing a performance bottleneck. In such a situation, CP4 lends itself to most easily evolving from a centralized to a distributed object-oriented database.

Our ongoing efforts involve the attempt to exploit software architectures as part of the generation of a URBS enforcement mechanism by ADAM. Specifically, we have been transitioning the UCLA and GEA approaches presented in Section 4.1 from C++ to Ada95. Once in Ada95, we can then exploit its process/tasking capabilities to examine the approaches presented in Section 4.2, with our likely initial focus on CP3. By developing a working prototype of a multi-processed object-oriented application with URBS, we can then explore the different levels of consistency and assurance that can be attained when changes to the URBS policy and/or application class library occur.

References

- [1] S. Demurjian and T.C. Ting, "The Factors that Influence Apropos Security Approaches for the Object-Oriented Paradigm", *Workshops in Computing*, Springer-Verlag, 1994.
- [2] S. Demurjian, M.-Y. Hu, and T.C. Ting, "Role-Based Access Control for Object-Oriented/C++ Systems", *Proc. of First ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, November 1995.
- [3] S. Demurjian, T.C. Ting, and M.-Y. Hu, "Security for Object-Oriented Databases, Systems, and Applications", in *Progress in Object-Oriented Databases*, J. Prater (ed.), Ablex, 1997.
- [4] S. Demurjian, T.C. Ting, M. Price, and M.-Y. Hu, "Extensible and Reusable Role-Based Object-Oriented Security", in *Database Security, X: Status and Prospects*, D. Spooner, P. Samarati, and R. Sandhu (eds.), Chapman Hall, 1997.
- [5] H. Ellis and S. Demurjian, "Object-Oriented Design and Analyses for Advanced Application Development - Progress Towards a New Frontier", *Proc. of the 21st Annual ACM Computer Science Conf.*, Feb. 1993.

- [6] W. Gotthard, et al., "System-Guided View Integration for Object-Oriented Databases", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 4, No. 1, Feb. 1992.
- [7] J. Hale, et al., "A Framework for High Assurance Security of Distributed Objects", *Proc. of 10th IFIP WG11.3 Working Conf. on Database Security*, Cumo, Italy, July 1996.
- [8] W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *Proc. of 1993 OOPSLA Conf.*, Oct. 1993.
- [9] M.-Y. Hu, S. Demurjian, and T.C. Ting, "User-Role Based Security Profiles for an Object-Oriented Design Model", in *Database Security, VI: Status and Prospects*, C. Landwehr and B. Thuraisingham (eds.), North-Holland, 1993.
- [10] M.-Y. Hu, S. Demurjian, and T.C. Ting, "Unifying Structural and Security Modeling and Analyses in the ADAM Object-Oriented Design Environment", in *Database Security, VIII: Status and Prospects*, J. Biskup, C. Landwehr, and M. Morgenstern (eds.), Elsevier Science, 1994.
- [11] D. Needham, S. Demurjian, K. El Guemhioui, T. Peters, P. Zemani, M. McMahon, H. Ellis "ADAM: A Language-Independent, Object-Oriented, Design Environment for Modeling Inheritance and Relationship Variants in Ada 95, C++, and Eiffel", *Proc. of 1996 TriAda Conf.*, Philadelphia, PA, December 1996.
- [12] H. Ossher, et al., "Subject-Oriented Composition Rules", *Proc. of 1995 OOPSLA Conf.*, Oct. 1995.
- [13] F. Rabitti, et al., "A Model of Authorization for Object-Oriented Database Systems", *Proc. of the Intl. Conf. on Extending Database Technology*, March 1988.
- [14] F. Rabitti, et al., "A Model of Authorization for Next Generation Database Systems", *ACM Trans. on Database Systems*, Vol. 16, No. 1, March 1991.
- [15] J. Richardson and P. Schwarz, "Aspects: Extending Objects to Support Multiple, Independent Roles", *Proc. of 1991 ACM SIGMOD Conf.*, May 1991.
- [16] M. Shaw and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Prentice-Hall, 1996.
- [17] D. Spooner, "The Impact of Inheritance on Security in Object-Oriented Database Systems", in *Database Security, II: Status and Prospects*, C. Landwehr (ed.), North-Holland, 1989.

Role-Based Administration of User-Role Assignment: The URA97 Model and its Oracle Implementation

Ravi Sandhu and Venkata Bhamidipati

Laboratory for Information Security Technology, ISSE Department, Mail Stop 4A4
George Mason University, Fairfax, VA 22033, sandhu@isse.gmu.edu

Abstract In role-based access control (RBAC) permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the roles' permissions. The principal motivation behind RBAC is to simplify administration. An appealing possibility is to use RBAC itself to manage RBAC, to further provide administrative convenience. In this paper we investigate one aspect of RBAC administration concerning assignment of users to roles. We define a role-based administrative model, called URA97 (user-role assignment '97), for this purpose and describe its implementation in the Oracle database management system. Although our model is quite different from that built into Oracle, we demonstrate how to use Oracle stored procedures to implement it.

1 INTRODUCTION

Role-based access control (RBAC) has recently received considerable attention as a promising alternative to traditional discretionary and mandatory access controls (see, for example, [FK92, FCK95, Gui95, GI96, MD94, HDT95, NO95, SCFY96, vSvdM94, YCS97]). In RBAC permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the roles' permissions. This greatly simplifies management of permissions. Roles are created for the various job functions in an organization and users are assigned roles based on their responsibilities and qualifications. Users can be easily reassigned from one role to another. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles as needed. Role-role relationships can be established to lay out broad policy objectives.

In large enterprise-wide systems the number of roles can be in the hundreds or thousands, and users can be in the tens or hundreds of thousands, maybe even

millions. Managing these roles and users, and their interrelationships is a formidable task that often is highly centralized and delegated to a small team of security administrators. Because the main advantage of RBAC is to facilitate administration of permissions, it is natural to ask how RBAC itself can be used to manage RBAC. We believe the use of RBAC for managing RBAC will be an important factor in the long-term success of RBAC. Decentralizing the details of RBAC administration without losing central control over broad policy is a challenging goal for system designers and architects.

As we will see there are many components to RBAC. RBAC administration is therefore multifaceted. In particular we can separate the issues of assigning users to roles, assigning permissions to roles, and assigning roles to roles to define a role hierarchy. These activities are all required to bring users and permissions together. However, in many cases, they are best done by different administrators (or administrative roles). Assigning permissions to roles is typically the province of application administrators. Thus a banking application can be implemented so credit and debit operations are assigned to a teller role, whereas approval of a loan is assigned to a managerial role. Assignment of actual individuals to the teller and managerial roles is a personnel management function. Assigning roles to roles has aspects of user-role assignment and role-permission assignment. Role-role relationships establish broad policy. Control of these relationships would typically be relatively centralized in the hands of a few security administrators.

In this paper we have focussed our attention exclusively on user-role assignment. We recognize that a comprehensive administrative model for RBAC must account for all three issues mentioned above, among others. However, user-role assignment is a particularly critical administrative activity. We feel it is the right one to focus on first.

In large systems user-role assignment is likely to be the first administrative function that is decentralized and delegated to users rather than system administrators. Assigning people to tasks is a normal managerial function. Assigning users to roles should be a natural part of assigning users to tasks. Empowering managers to do this routinely is one way of making security an enabling user-friendly technology rather than an intrusive and cumbersome nuisance as it all too often turns out to be. A manager who can assign a user to perform certain tasks should not have to ask someone else to enroll this user in appropriate roles. This should happen transparently and conveniently.

A user-role assignment model can also be used for managing user-group assignment and therefore has applicability beyond RBAC. The difference between roles and groups was hotly debated at the First ACM Workshop on RBAC [San97b]. Workshop attendees arrived at the consensus that a group is a named collection of users (and possibly other groups). Groups serve as a convenient shorthand notation for collections of users and that is the main motivation for introducing them. Roles are similar to groups in that they can serve as a shorthand for collections of users, but they go beyond groups in also serving as a shorthand for a collection of permissions. Assigning users to roles or users to groups are therefore essentially the same function. Assigning permissions to roles and permissions to groups, on the other hand, can have rather different characteristics. We need not get into this latter issue here since our focus is on user-role, or equivalently user-group, assignment.

In this paper we propose a model for the assignment of users to roles by means of administrative roles and permissions. We call our model URA97 (user-role assignment '97). URA97 imposes strict limits on individual administrators regarding which users can be assigned to which roles. We then describe an implementation of URA97 in the Oracle database management system [KL95, Feu95]. Oracle's administrative model for user-role assignment is very different from URA97. Nevertheless, we show how to use Oracle's stored procedures to implement URA97.

The principal contribution of URA97 is to provide a concrete example of what is meant by role-based administration of user-role assignment. Another central contribution of this paper is to demonstrate that an existing popular product, namely Oracle, provides the necessary base mechanisms and extensibility to program the behavior of URA97. URA97 is defined in context of the family of RBAC96 family of models due to Sandhu et al [SCFY96]. How-

ever, it applies to almost any RBAC model, including [FCK95, Gui95, GI96, HDT95, NO95], because user-role assignment is a basic administrative feature which will be required in any RBAC model.

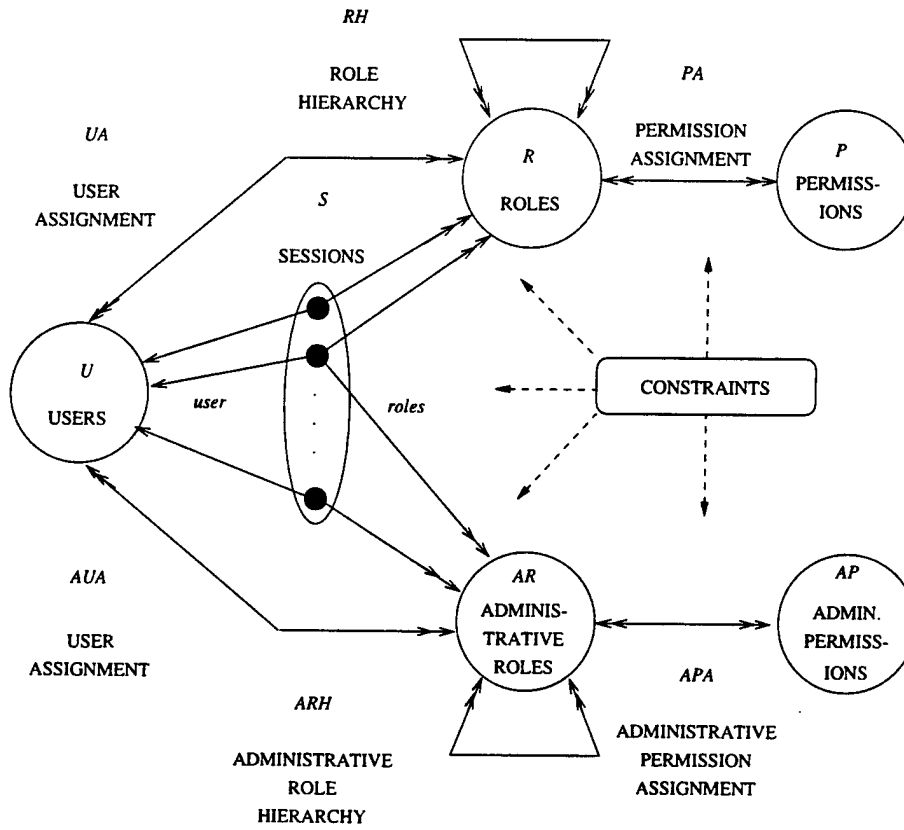
The rest of this paper is organized as follows. We begin by reviewing the RBAC96 family of models in section 2. In section 3 we define the administrative model called URA97 for user-role assignment which itself is role-based. This is followed by a quick review of relevant RBAC features of Oracle in section 4. Our implementation of URA97 in Oracle is described in section 5. Section 6 concludes the paper.

2 THE RBAC96 MODELS

A general family of RBAC models called RBAC96 was defined by Sandhu et al [SCFY96]. Figure 1 illustrates the most general model in this family. For simplicity we use the term RBAC96 to refer to the family of models as well as its most general member.

The top half of figure 1 shows (regular) roles and permissions that regulate access to data and resources. The bottom half shows administrative roles and permissions. Intuitively, a user is a human being or an autonomous agent, a role is a job function or job title within the organization with some associated semantics regarding the authority and responsibility conferred on a member of the role, and a permission is an approval of a particular mode of access to one or more objects in the system or some privilege to carry out specified actions. Roles are organized in a partial order \geq , so that if $x \geq y$ then role x inherits the permissions of role y . Members of x are also implicitly members of y . In such cases, we say x is senior to y . Each session relates one user to possibly many roles. The idea is that a user establishes a session and activates some subset of roles that he or she is a member of (directly or indirectly by means of the role hierarchy).

Motivation and discussion about various design decisions made in developing this family of models is given in [SCFY96, San97a]. It is worth emphasizing that RBAC96 distinguishes roles and permissions from administrative roles and permissions respectively, where the latter are used to manage the former. How are administrative permissions and roles managed in turn? One could consider a second level of administrative roles and permissions to manage the first level ones and so on. We feel such a progression of administration is unnecessary. Administration of administrative roles and permissions is under control



- U , a set of users
 R and AR , disjoint sets of (regular) roles and administrative roles
 P and AP , disjoint sets of (regular) permissions and administrative permissions
 S , a set of sessions
- $UA \subseteq U \times R$, user to role assignment relation
 $AUA \subseteq U \times AR$, user to administrative role assignment relation
- $PA \subseteq P \times R$, permission to role assignment relation
 $APA \subseteq AP \times AR$, permission to administrative role assignment relation
- $RH \subseteq R \times R$, partially ordered role hierarchy
 $ARH \subseteq AR \times AR$, partially ordered administrative role hierarchy
 (both hierarchies are written as \geq in infix notation)
- $user : S \rightarrow U$, maps each session to a single user (which does not change)
 $roles : S \rightarrow 2^{R \cup AR}$ maps each session s_i to a set of roles and administrative roles $roles(s_i) \subseteq \{r \mid (\exists r' \geq r)[(user(s_i), r') \in UA \cup AUA]\}$ (which can change with time)
 session s_i has the permissions $\bigcup_{r \in roles(s_i)} \{p \mid (\exists r'' \leq r)[(p, r'') \in PA \cup APA]\}$
- there is a collection of constraints stipulating which values of the various components enumerated above are allowed or forbidden.

Figure 1: Summary of the RBAC96 Model

of the chief security officer or delegated in part to administrative roles.

3 THE URA97 MODEL

RBAC has many components as described in the previous section. Administration of RBAC involves control over each of these components including creation and deletion of roles, creation and deletion of permissions, assignment of permissions to roles and their removal, creation and deletion of users, assignment of users to roles and their removal, definition and maintenance of the role hierarchy, definition and maintenance of constraints and all of these in turn for administrative roles and permissions. A comprehensive administrative model would be quite complex and difficult to develop in a single step.

Fortunately administration of RBAC can be partitioned into several areas for which administrative models can be separately and independently developed to be later integrated. In particular we can separate the issues of assigning users to roles, assigning permissions to roles and defining the role hierarchy. In many cases, these activities would be best done by different administrators. Assigning permissions to roles is typically the province of application administrators. Thus a banking application can be implemented so credit and debit operations are assigned to a teller role, whereas approval of a loan is assigned to a managerial role. Assignment of actual individuals to the teller and managerial roles is a personnel management function. Design of the role hierarchy relates to design of the organizational structure and is the function of a chief security officer under guidance of a chief information officer.

In this paper our focus is exclusively on user-role assignment. As discussed in section 1 this is likely to be the first and most widely decentralized administrative task in RBAC. In the RBAC96 framework of figure 1 control of UA is vested in the administrative roles AR . For simplicity we limit our scope to assignment of users to regular roles. Assignment of users to administrative roles is centralized under the chief security officer. In general the chief security officer has complete control over all aspects of RBAC96.

In the rest of this section we develop a model called URA97 in which RBAC is used to manage user-role assignment. We define URA97 in two steps dealing with granting a user membership in a role and revoking a user's membership. URA97 is deliberately designed

to have a very narrow scope. For example creation of users and roles is outside its scope. In spite of its simplicity URA97 is quite powerful and goes much beyond existing administrative models for user-role assignment, such as the one implemented in Oracle. It is also applicable beyond RBAC to user-group assignment.

3.1 URA97 Grant Model

In the simplest case user-role assignment can be completely centralized in a single chief security officer role. This is readily implemented in existing systems such as Oracle. However, this simple approach does not scale to large systems. Clearly it is desirable to decentralize user-role assignment to some degree.

In several systems, including Oracle, it is possible to designate a role, say, junior security officer (JSO) whose members have administrative control over one or more regular roles, say, A, B and C. Thus limited administrative authority is delegated to the JSO role. Unfortunately these systems typically allow the JSO role to have complete control over roles A, B and C. A member of JSO can not only add users to A, B and C but also delete users from these roles and add and delete permissions. Moreover, there is no control on which users can be added to the A, B and C roles by JSO members. Finally, JSO members are allowed to assign A, B and C as junior to any role in the existing hierarchy (so long as this does not introduce a cycle). All this is consistent with classical discretionary thinking whereby member of JSO are effectively designated as "owners" of the A, B and C roles, and therefore are free to do whatever they want to these roles.

In URA97 our goal is to impose restrictions on which users can be added to a role by whom, as well as to clearly separate the ability to add and remove users from other operations on the role. The notion of a prerequisite condition is a key part of URA97.

Definition 1 A **prerequisite condition** is a boolean expression using the usual \wedge and \vee operators on terms of the form x and \bar{x} where x is a regular role (i.e., $x \in R$). A prerequisite condition is evaluated for a user u by interpreting x to be true if $(\exists x' \geq x)(u, x') \in UA$ and \bar{x} to be true if $(\forall x' \geq x)(u, x') \notin UA$. For a given set of roles R let CR denotes all possible prerequisite conditions that can be formed using the roles in R . \square

In the trivial case a prerequisite condition can be a tautology which is always true. The simplest non-

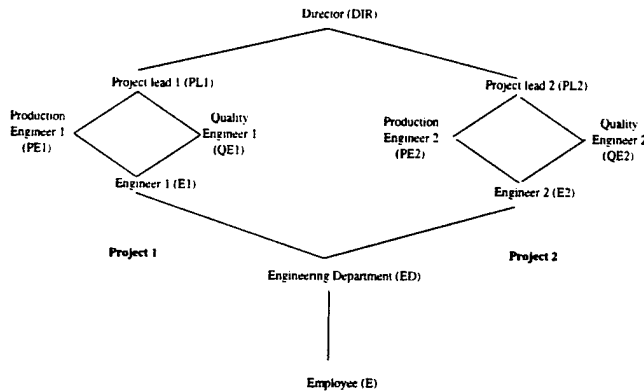


Figure 2: An Example Role Hierarchy

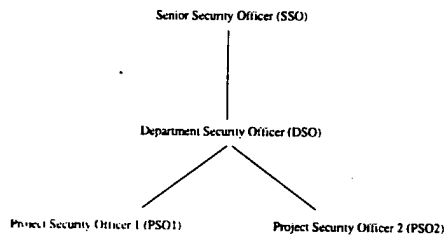


Figure 3: An Example Administrative Role Hierarchy

trivial case of a prerequisite condition is test for membership in a single role, in which situation that single role is called a prerequisite role.

User-role assignment is authorized in URA97 by the following relation.

Definition 2 The URA97 model controls user-role assignment by means of the relation $can_assign \subseteq AR \times CR \times 2^R$. \square

The meaning of $can_assign(x, y, \{a, b, c\})$ is that a member of the administrative role x (or a member of an administrative role that is senior to x) can assign a user whose current membership, or non-membership, in regular roles satisfies the prerequisite condition y to be a member of regular roles a, b or c .¹

To appreciate the motivation behind the can_assign relation consider the role hierarchy of figure 2 and the administrative role hierarchy of figure 3. Figure 2

¹User-role assignment is subject to constraints, such as mutually exclusive roles or maximum cardinality, that may be imposed. The assignment will succeed if and only if it is authorized by can_assign and it satisfies all relevant constraints.

shows the regular roles that exist in a engineering department. There is a junior-most role E to which all employees in the organization belong. Within the engineering department there is a junior-most role ED and senior-most role DIR. In between there are roles for two projects within the department, project 1 on the left and project 2 on the right. Each project has a senior-most project lead role (PL1 and PL2) and a junior-most engineer role (E1 and E2). In between each project has two incomparable roles, production engineer (PE1 and PE2) and quality engineer (QE1 and QE2).

Figure 2 suffices for our purpose but this structure can, of course, be extended to dozens and even hundreds of projects within the engineering department. Moreover, each project could have a different structure for its roles. The example can also be extended to multiple departments with different structure and policies applied to each department.

Figure 3 shows the administrative role hierarchy which co-exists with figure 2. The senior-most role is the senior security officer (SSO). Our main interest is in the administrative roles junior to SSO. These consist of two project security officer roles (PSO1 and PSO2) and a department security officer (DSO) role with the relationships illustrated in the figure.

3.1.1 Prerequisite Roles

For sake of illustration we define the can_assign relation shown in table 1(a). This example has the simplest prerequisite condition of testing membership in a single role known as the prerequisite role.

The PSO1 role has partial responsibility over project 1 roles. Let Alice be a member of the PSO1 role and Bob a member of the ED role. Alice can assign Bob to any of the E1, PE1 and QE1 roles, but not to the PL1 role. Also if Charlie is not a member of the ED role, then Alice cannot assign him to any project 1 role. Hence, Alice has authority to enroll users in the E1, PE1 and QE1 roles provided these users are already members of ED. Note that if Alice assigns Bob to PE1 he does not need to be explicitly assigned to E1, since E1 permissions will be inherited via the role hierarchy. The PSO2 role is similar to PSO1 but with respect to project 2. The DSO role inherits the authority of PSO1 and PSO2 roles but can further add users who are members of ED to the PL1 and PL2 roles. The SSO role can add users who are in the E role to the ED role, as well as add users who are in the ED role to the DIR role. This ensures that even

Admin. Role	Prereq. Role	Role Set
PSO1	ED	{E1, PE1, QE1}
PSO2	ED	{E2, PE2, QE2}
DSO	ED	{PL1, PL2}
SSO	E	{ED}
SSO	ED	{DIR}

(a) Subset Notation

Admin. Role	Prereq. Role	Role Range
PSO1	ED	[E1, PL1)
PSO2	ED	[E2, PL2)
DSO	ED	(ED, DIR)
SSO	E	[ED, ED]
SSO	ED	(ED, DIR]

(b) Range Notation

Table 1: *can-assign* with Prerequisite Roles

the SSO must first enroll a user in the ED role before that user is enrolled in a role senior to ED. This is a reasonable specification for *can-assign*. There are, of course, lots of other equally reasonable specifications in this context. This is a matter of policy decision and our model provides the necessary flexibility.

In general, one would expect that the role being assigned is senior to the role previously required of the user. That is, if we have *can-assign*(a, b, C) then b is junior to all roles $c \in C$. We believe this will usually be the case, but we do not require it in the model. This allows URA97 to be applicable to situations where there is no role hierarchy or where such a constraint may not be appropriate.

The notation of table 1(a) has benefited from the administrative role hierarchy. Thus for the DSO we have specified the role set as {PL1, PL2} and the other values are inherited from PSO1 and PSO2. Similarly for the SSO. Nevertheless explicit enumeration of the role set is unwieldy, particularly if we were to scale up to dozens or hundreds of projects in the department. Moreover, explicit enumeration is not resilient with respect to changes in the role hierarchy. Suppose a third project is introduced in the department, with roles E3, PE3, QE3, PL3 and PSO3 analogous to corresponding roles for projects 1 and 2. We can add the following row to table 1(a).

Admin. Role	Prereq. Role	Role Set
PSO3	ED	{E3, PE3, QE3}

This is a reasonable change to require when the new project and its roles are introduced into the regular and administrative role hierarchies. However, we also need to modify the row for DSO in table 1(b) to include PL3.

3.1.2 Range Notation

Consider instead the range notation illustrated in table 1(b). Table 1(b) shows the same role sets as table 1(a) but defines these sets by identifying a range within the role hierarchy of figure 1(a) by means of the familiar closed and open interval notation.

Definition 3 Role sets are specified in the URA97 model by the notation below

$$\begin{aligned}
[x, y] &= \{r \in R \mid x \geq r \wedge r \geq y\} \\
(x, y] &= \{r \in R \mid x > r \wedge r \geq y\} \\
[x, y) &= \{r \in R \mid x \geq r \wedge r > y\} \\
(x, y) &= \{r \in R \mid x > r \wedge r > y\}
\end{aligned}$$

□

This notation is resilient to modifications in the role hierarchy such as addition of a third project which requires addition of the following row to table 1(b).

Admin. Role	Prereq. Role	Role Range
PSO3	ED	[E3, PL3)

No other change is required since the [ED, DIR) range specified for the DSO will automatically pick up PL3.

The range notation is, of course, not resilient to all changes in the role hierarchy. Deletion of one of the end points of a range can leave a dangling reference and an invalid range. Standard techniques for ensuring referential integrity would need to be applied when modifying the range hierarchy. Changes to role-role relationships could also cause a range to be drastically different from its original meaning. Nevertheless the range notation is much more convenient than explicit enumeration. There is also no loss of generality in adopting the range notation since every set of roles can be expressed as a union of disjoint ranges.

Strictly speaking the two specifications of table 1(a) and 1(b) are not precisely identical. In table 1(a) the DSO role is explicitly authorized to enroll users in PL1 and PL2, and inherits the ability to enroll users in other project 1 and 2 roles from PSO1 and PSO2. On

Admin. Role	Prereq. Condition	Role Range
PSO1	ED	[E1, E1]
PSO1	$ED \wedge \overline{QE1}$	[PE1, PE1]
PSO1	$ED \wedge \overline{PE1}$	[QE1, QE1]
PSO1	$PE1 \wedge QE1$	[PL1, PL1]
PSO2	ED	[E2, E2]
PSO2	$ED \wedge \overline{QE2}$	[PE2, PE2]
PSO2	$ED \wedge \overline{PE2}$	[QE2, QE2]
PSO2	$PE2 \wedge QE2$	[PL2, PL2]
DSO	ED	(ED, DIR)
SSO	E	[ED, ED]
SSO	ED	(ED, DIR)

Table 2: *can-assign* with Prerequisite Conditions

the other hand, in table 1(b) the DSO role is explicitly authorized to enroll users in all project 1 and 2 roles. As it stands the net effect is the same. However, if modifications are made to the role hierarchy or to the PSO1 or PSO2 authorizations the effect can be different. The DSO authorization in table 1(a) can be replaced by the following row to make table 1(a) more nearly identical to table 1(b).

Admin. Role	Prereq. Role	Role Set
DSO	ED	{E1, PE1, QE1, PL1, E2, PE2, QE2, PL2}

Now even if the PSO1 and PSO2 roles of table 1(a) are modified respectively to the role sets {E1} and {E2}, the DSO role will still retain administrative authority over all project 1 and project 2 roles. Of course, explicit and implicit specifications will never behave exactly identically under *all* circumstances. For instance, introduction of a new project 3 will exhibit differences as discussed above. Conversely, the DSO authorization in table 1(b) can be replaced by the following rows to make table 1(b) more nearly identical to table 1(a).

Admin. Role	Prereq. Role	Role Range
DSO	ED	[PL1, PL1]
DSO	ED	[PL2, PL2]

There is an analogous situation with the SSO role in tables 1(a) and 1(b). Clearly, we must anticipate the impact of future changes when we specify the *can-assign* relation.

3.1.3 Prerequisite Conditions

An example of *can-assign* which uses prerequisite conditions rather than prerequisite roles is shown in table 2. The authorizations for PSO1 and PSO2 have been changed relative to table 1.

Let us consider the PSO1 tuples (analysis for PSO2 is exactly similar). The first tuple authorizes PSO1 to assign users with prerequisite role ED into E1. The second one authorizes PSO1 to assign users with prerequisite condition $ED \wedge \overline{QE1}$ to PE1. Similarly, the third tuple authorizes PSO1 to assign users with prerequisite condition $ED \wedge \overline{PE1}$ to QE1. Taken together the second and third tuples authorize PSO1 to put a user who is a member of ED into one but not both of PE1 and QE1. This illustrates how mutually exclusive roles can be enforced by URA97. PE1 and QE1 are mutually exclusive with respect to the power of PSO1. However, for the DSO and SSO these are not mutually exclusive. Hence, the notion of mutual exclusion is a relative one in URA97. The fourth tuple authorizes PSO1 to put a user who is a member of both PE1 and QE1 into PL1. Of course, a user could have become a member of both PE1 and QE1 only by actions of a more powerful administrator than PSO1.

3.2 URA97 Revoke Model

We now turn to consideration of the URA97 revoke model. The objective is to define a revoke model that is consistent with the philosophy of RBAC. This causes us to depart from classical discretionary approaches to revocation.

In the classical discretionary approach to revocation there are at least two issues that introduce complexity and subtlety [GW76, Fag78]. Suppose Alice grants Bob some permission P. This is done at Alice's discretion because Alice is either the owner of the object to which P pertains or has been granted administrative authority on P by the actual owner. Alice can later revoke P from Bob. Now suppose Bob has received permission P from Alice and from Charlie. If Alice revokes her grant of P to Bob he should still continue to retain P because of Charlie's grant. A related issue is that of cascading revokes. Suppose Charlie's grant was in turn obtained from Alice, perhaps Bob's permission should end up being revoked by Alice's action. Or perhaps it should not, because Alice only revoked her direct grant to Bob but not the indirect one via Charlie which really occurred at Charlie's discretion. A considerable literature has developed examining the subtleties that arise, espe-

cially when hierarchical groups and negative permissions or denials are brought into play (see, for example, [Lun88, BSJ93, FWF95, GSF91, RBKW91]).

The RBAC approach to authorization is quite different from the traditional discretionary one. In RBAC users are made members of roles because of their job function or task assignment in the interest of the organization. Granting of membership in a role is specifically not done at the grantor's whim. Suppose Alice makes Bob a member of a role X . In URA97 this happens because Alice is assigned suitable administrative authority over X via some administrative role Y and Bob is eligible for membership in X due to Bob's existing role memberships (and non-memberships) satisfying the prerequisite condition. Moreover, there are some organizational circumstances which cause Alice to grant Bob this membership. It is not merely being done at Alice's personal fancy. Now if at some later time Alice is removed from the administrative role Y there is clearly no reason to also remove Bob from X . A change in Alice's job function should not necessarily undo her previous grants. Presumably some other administrator, say Dorothy, will take over Alice's responsibility. Similarly, suppose Alice and Charlie both grant membership to Bob in X . At some later time Bob is reassigned to some other project and no longer needs to be a member of role X . It is not material whether Alice or Charlie or both or Dorothy revokes Bob's membership. Bob's membership in X is being revoked due to a change in organizational circumstances.

To summarize, in classical discretionary access control the source (direct or indirect) of a permission and the identity of the revoker is typically taken into account in interpreting the revoke operation.² These issues do not arise in the same way for revocation of user-role assignment in RBAC. However, there are related subtleties that arise in RBAC concerning the interaction between granting and revocation of user-role membership and the role hierarchy. We will illustrate these in a moment.

3.2.1 The Can-Revoke Relation

We now introduce our notation for authorizing revocation.

Definition 4 The URA97 model controls user-role revocation by means of the relation *can-revoke* \subseteq

²This is true more in theory than practice, because many commercial products opt for a simpler semantics than implied by a strict owner-based discretionary viewpoint.

$AR \times 2^R$. □

The meaning of *can-revoke*(x, Y) is that a member of the administrative role x (or a member of an administrative role that is senior to x) can revoke membership of a user from any regular role $y \in Y$. Y is specified using the range notation of definition 3. We say Y defines the *range of revocation*.

3.2.2 Weak Revocation

The revocation operation in URA97 is said to be **weak** because it applies only to the role that is directly revoked. Suppose Bob is a member of PE1 and E1. If Alice revokes Bob's membership from E1, he continues to be a member of the senior role PE1 and therefore can use the permissions of E1.

To make the notion of weak revocation precise we introduce the following terminology. Recall that UA is the user assignment relation.

Definition 5 Let us say a user U is an *explicit member* of role x if $(U, x) \in UA$, and that U is an *implicit member* of role x if for some $x' > x$, $(U, x') \in UA$. □

Note that a user can simultaneously be an explicit and implicit member of a role.

Weak revocation has an impact only on explicit membership. It has the straightforward meaning stated below.

Definition 6 [Weak Revocation Algorithm]

1. Let Alice have a session with administrative roles $A = \{a_1, a_2, \dots, a_k\}$, and let Alice try to weakly revoke Bob from role x .
2. If Bob is not an explicit member of x this operation has no effect, otherwise there are two cases.

- (a) There exists a *can-revoke* tuple (b, Y) such that there exists $a_i \in A, a_i \geq b$ and $x \in Y$.

In this case Bob's explicit membership in x is revoked.

- (b) There does not exist a *can-revoke* tuple as identified above.

In this case the weak revoke operation has no effect. □

Admin. Role	Role Range
PSO1	[E1, PL1)
PSO2	[E2, PL2)
DSO	(ED, DIR)
SSO	[ED, DIR]

Table 3: Example of *can-revoke*

3.2.3 Strong Revocation

Strong revocation in URA97 requires revocation of both explicit and implicit membership. Strong revocation of U's membership in x requires that U be removed not only from explicit membership in x , but also from explicit (or implicit) membership in all roles senior to x . Strong revocation therefore has a cascading effect upwards in the role hierarchy. However, strong revocation in URA97 takes effect only if all implied revocations upward in the role hierarchy are within the revocation range of the administrative roles that are active in a session.

In other words strong revocation is equivalent to a series of weak revocations. Although it is theoretically redundant, strong revocation is a useful and convenient operation for administrators. It is much better for the system to figure out what weak revocations need to be carried out to achieve strong revocation, rather than leave it to administrators to determine this.

Let us consider the example of *can-revoke* shown in table 3 and interpret it in context of the hierarchies of figures 2 and 3. Let Alice be a member of PSO1, and let this be the only administrative role she has. Alice is authorized to strongly revoke membership of users from roles E1, PE1 and QE1. Table 4(a) illustrates whether or not Alice can strongly revoke membership of a user from role E1. The effect of Alice's strong revocation of each of these users from E1 is shown in table 4(b). Alice is not allowed to strongly revoke Dave and Eve from E1 because they are members of senior roles outside the scope of Alice's revoking authority. If Alice was assigned to the DSO role she could strongly revoke Dave from E1 but still would not be able to strongly revoke Eve's membership in E1. In order to strongly revoke Eve from E1, Alice needs to be in the SSO role.

The algorithm for strong revocation is stated in terms of weak revocation as follows.

Definition 7 [Strong Revocation Algorithm]

1. Let Alice have a session with administrative roles $A = \{a_1, a_2, \dots, a_k\}$, and let Alice try to strongly revoke Bob from role x .
2. Find all roles $y \geq x$ and Bob is a member of y .
3. Weak revoke Bob from all such y as if Alice did this weak revoke.
4. If any of the weak revokes fail then Alice's strong revoke has no effect otherwise all weak revokes succeed.

An alternate approach would be to do only those weak revokes that succeed and ignore the rest. We decided to go with a cleaner all-or-nothing semantics in URA97.

So far we have looked at the cascading of revocation upward in the role hierarchy. There is a downward cascading effect that also occurs. Consider Bob in our example who is a member of E1 and PE1. Suppose further that Bob is an explicit member of PE1 and thereby an implicit member of E1. What happens if Alice revokes Bob from PE1? If we remove (Bob, PE1) from the UA relation, Bob's implicit membership in E1 will also be removed. On the other hand if Bob is an explicit member of PE1 and also an explicit member of E1 then Alice's revocation of Bob from PE1 does not remove him from E1. The revoke operations we have defined in URA97 have the following effect.

Property 1. Implicit membership in a role a is dependent on explicit membership in some senior role $b > a$. Therefore when explicit membership of a user is revoked from b , implicit membership is also automatically revoked on junior role a unless there is some other senior role $c > a$ in which the user continues to be an explicit member. (This will require $b \neq c$.)

Note that our examples of *can-assign* in table 1(b) and *can-revoke* in table 3 are complementary in that each administrative role has the same range for adding users and removing users from roles. Although this would be a common case we do not impose it as a requirement on our model.

3.3 Summary of URA97

URA97 controls user-role assignment by means of the relation $can_assign \subseteq AR \times CR \times 2^R$. Role

User	E1	PE1	QE1	PL1	DIR	Alice can revoke user from E1
Bob	Yes	Yes	No	No	No	Yes
Cathy	Yes	Yes	Yes	No	No	Yes
Dave	Yes	Yes	Yes	Yes	No	No
Eve	Yes	Yes	Yes	Yes	Yes	No

(a) Prior to strong revocation

User	E1	PE1	QE1	PL1	DIR	Alice revoke user from E1
Bob	No	No	No	No	No	removed from E1, PE1
Cathy	No	No	No	No	No	removed from E1, PE1, QE1
Dave	Yes	Yes	Yes	Yes	Yes	no effect
Eve	Yes	Yes	Yes	Yes	Yes	no effect

(b) After strong revocation

Table 4: Example of Strong Revocation

sets are specified using the range notation of definition 3. Assignment has a simple behavior whereby $\text{can-assign}(a, b, C)$ authorizes a session with an administrative role $a' \geq a$ to enroll any user who satisfies the prerequisite condition b into any role $c \in C$. The prerequisite condition is a boolean expression using the usual \wedge and \vee operators on terms of the form x and \bar{x} respectively denoting membership and non-membership regular role x .

Revocation is controlled in URA97 by the relation $\text{can-revoke} \subseteq AR \times 2^R$. Weak revocation applies only to explicit membership in a single role as per the algorithm of definition 6. Strong revocation cascades upwards in the role hierarchy as per the algorithm of definition 7. In both cases revocation cascades downwards as noted in property 1.

4 ORACLE RBAC FEATURES

The Oracle database management system [KL95, Feu95] provides support for RBAC including support for hierarchical roles. However, Oracle does not directly support the URA97 model. In particular, Oracle has a strong discretionary flavor to its administrative model for user-role assignment and revocation. Also the Oracle revocation model is similar to our weak revoke and does not cascade revocation upwards in the role hierarchy like our strong revoke does. This is reasonable given Oracle's discretionary orien-

tation. Nevertheless, we will see in the next section how it is possible to use Oracle's stored procedures to implement URA97. In this section we briefly review relevant features of Oracle access control.

4.1 Privileges

Oracle has two kinds of privileges, system privileges and object privileges. System privileges authorize actions on a particular type of object for example create table, create user, etc. There are over 60 distinct system privileges. Object privileges authorize actions on a specific object (table, view, procedure, package etc.). Typical examples of object privileges are select rows from a table, delete rows, execute procedures etc.

Who can grant or revoke privileges from users or roles? The answer depends on various issues such as whether it is a system or an object privilege, and whether the object is owned by the user, etc. In order to grant or revoke a system privilege the user should have the admin option on that privilege or the user should have GRANT ANY PRIVILEGE system privilege. In order to grant or revoke an object privilege a user should own that particular object or the user should have grant option on the object if it is owned by someone else.

4.2 Roles in Oracle

Oracle provides roles (from Oracle 7.0 onwards) for ease of management of privilege assignment. System and object privileges can be granted to a role. A role can be granted to any other role (circular granting is not allowed). Any role can be granted to any user in the database. A role can either be enabled or disabled during a session. This includes both explicit and implicit roles that a user is a member of. Enabling a role will implicitly enable all the roles granted to it directly or transitively. The system privileges related to role management are `CREATE_ROLE`, `GRANT_ANY_ROLE`, `DROP_ROLE`, and `DROP_ANY_ROLE`.

Information about privileges assigned to a role can be obtained from Oracle's built-in views `ROLE_SYS_PRIVILEGES`, `ROLE_TAB_PRIVILEGES`, and `ROLE_ROLE_PRIVS`. When a regular user performs query on these views these views only show information pertaining to the roles granted to that user. However, the Oracle internal user `SYS` will see information about all the roles through these views. The view `SESSION_ROLES` provides information about roles that are enabled in a session. The view `ROLE_ROLE_PRIVS` shows information about which roles are directly assigned to another role. Roles inherited transitively are not shown. For example, if role C was granted to role B and role B to role A the `ROLE_ROLE_PRIVS` view will show that B has been granted to A and C to B, but will not show the implied transitive C to A grant.

4.3 Procedures, Functions and Packages

Oracle provides a programmatic approach to manipulate database information using procedural schema objects called PL/SQL (Procedural Language/SQL) program units. Procedures, functions and packages are different types of PL/SQL objects. PL/SQL extends the capabilities of SQL by providing some programming language features such as conditional statements, loops etc. Procedures are also referred to as stored procedures.

A procedure is a collection of instructions which can be grouped together and are performed on database objects to add, modify or delete database information. In order to create a procedure a user should have the `CREATE_PROCEDURE` system privilege. A procedure can be executed by a user who owns it or by a user who has execute privileges on it.

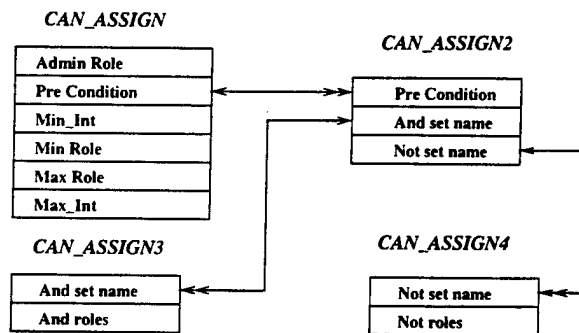


Figure 4: Entity-Relation Diagram for *can-assign*

A stored procedure runs with the privileges of the user who owns it and not the user who is executing it. This feature gives great flexibility in enforcing security. For example suppose we want a user to perform some operations on a database but we do not want to grant privileges explicitly. Then one can write a procedure embedded with necessary operations, and grant execute privileges on the procedure to the user.³

Functions are very similar to procedures. The only difference between a function and a procedure is that a procedure call is a PL/SQL statement itself, while functions are called as part of an expression. A function always returns a value when it is called.

Packages are PL/SQL constructs that store related objects together. A package is essentially a named declarative section. It can contain procedures, functions, variables etc. A package consists of two parts, the specification part and body, stored separately in the data dictionary. The package specification, also known as package header, contains the information about the contents of the package. The package body contains code for the subprograms declared in the header.

5 IMPLEMENTING URA97 IN ORACLE

To implement URA97 we define Oracle relations which encode the *can-assign* and *can-revoke* relations of URA97. The *can-assign* relation of URA97 is imple-

³The privileges that are referenced in a procedure should have been explicitly granted to the user who owns the procedure. Privileges obtained by the owner via a role cannot be referenced in a procedure.

AR	PC	Min	Min_Role	Max_Role	Max
PSO1	C1	[E1	E1]
PSO1	C2	[PE1	PE1]
PSO1	C3	[QE1	QE1]
PSO1	C4	[PL1	PL1]
...

(a) *can-assign*

PC	and_set	not_set
C1	ASET1	null
C2	ASET2	NSET2
C3	ASET3	NSET3
C4	ASET4	null
...

(b) *can-assign2*

and_set	and_roles
ASET1	ED
ASET2	ED
ASET3	ED
ASET4	PE1
ASET4	QE1
...	...

(c) *can-assign3*

not_set	not_roles
NSET2	QE1
NSET3	PE1
...	...

(d) *can-assign4*

Table 5: Oracle *can-assign* Relations for PSO1 from Table 2

AR	PC	Min	Min_Role	Max_Role	Max
SO1	C1
...

(a) *can-assign*

PC	and_set	not_set
C1	ASET1	NSET1
C1	ASET2	NSET2
...

(b) *can-assign2*

and_set	and_roles
ASET1	A
ASET1	D
ASET2	B
...	...

(c) *can-assign3*

not_set	not_roles
NSET1	E
NSET2	F
NSET2	D
...	...

(d) *can-assign4*

Table 6: Oracle *can-assign* Relations for Prerequisite Condition $(A \wedge D \wedge \bar{E}) \vee (B \wedge \bar{D} \wedge \bar{F})$

AR	Min	Min_Role	Max_Role	Max
PSO1	[E1	PL1)
PSO2	[E2	PL2)
DSO	(ED	DIR)
SSO	[ED	DIR]

Table 7: Oracle *can-revoke* Relation

mented in Oracle as per the entity-relation diagram of figure 4. We assume that the prerequisite condition is converted into disjunctive normal form using standard techniques. Disjunctive normal form has the following structure.

$$(\dots \wedge \dots \wedge \dots \wedge \dots) \vee (\dots \wedge \dots \wedge \dots \wedge \dots) \vee \dots \vee (\dots \wedge \dots \wedge \dots \wedge \dots)$$

Each \dots is a positive literal x or a negated literal \bar{x} . Each group $(\dots \wedge \dots \wedge \dots \wedge \dots)$ is called a disjunct. For a given prerequisite condition *can-assign2* has a tuple for each disjunct. All positive literals of a single disjunct are in *can-assign3*, while negated literals are in *can-assign4*.

The four PSO1 tuples of table 2 are represented by this scheme as shown in table 5. The prerequisite conditions in this case all have a single disjunct. An example with multiple disjuncts is shown in table 6.

The *can-revoke* relation of URA97 is represented by a single Oracle relation. For example table 3 is represented as shown in table 7.

The *can-assign*, *can-assign*, *can-assign*, *can-assign*, and *can-revoke* relations are owned by the DBA who also decides what their content should be. In addition we have three accompanying procedures and a package to support these. There is one procedure each for assigning a user to a role, doing a weak revoke of membership and doing a strong revoke of membership, respectively as follows.

- ASSIGN
- WEAK_REVOKE
- STRONG_REVOKE

Execute privilege on these procedures is given to all administrative roles. We achieve this by introducing a junior-most administrative role, say GSO (generic security officer), and assigning it the permission to execute these procedures.

These relations and accompanying procedures and packages are owned by the DBA. Our implementation also maintains an audit relation which keeps a log of all attempted assignment and revoke operations and their outcome. The audit relation is also owned by the DBA.

Oracle does not provide convenient primitives for testing whether or not a user is an implicit member of a particular role. Testing explicit membership is straightforward since explicit membership is encoded as a tuple in Oracle's system relations. To test implicit membership, however, we need to chase the role hierarchy. Oracle also does not provide direct support for enumerating roles in a range set. We built a PL/SQL package to support these requirements and assist in writing our stored procedures, as discussed below.

One of the problem we encountered was the inability for a stored procedure to determine which roles have been turned on in a given session. Let us say Alice is a member of the SSO role in our running example. This gives her implicit membership in all administrative roles. In RBAC96 Alice should be able to decide which, if any, of these administrative roles to turn on in a given session. Oracle allows turning roles on and off in this manner. Unfortunately when Alice invokes a stored procedure there is no means to determine from within the stored procedure as to which roles Alice has turned on in that particular session. This is a major obstacle in implementing URA97 in Oracle. In fact this problem arises for all kinds of extensions that could be proposed for Oracle RBAC via stored procedures. The problem arises because when a stored procedure is created the code and execution path of queries in the procedure are compiled and stored within the database. So when a stored procedure is called it is not possible to determine which roles are turned on in that session because the Oracle SESSION_ROLES view is based on the current session running and it's execution path can not be predefined. The standard Oracle technique for finding the roles of a session returns the empty set if invoked within a stored procedure. We are told that Oracle is aware of this problem and may have a fix in future releases. However, in the interim, we can overcome this problem by using a suitable Oracle GUI front end tool like Oracle Forms or by using Oracle Call Interface (an API tool). In both cases we can use IS_ROLE_ENABLED function to determine whether a role is enabled and SET_ROLE procedure for enabling a role. These functionalities are part of an Oracle Package called DBMS_SESSION. Unlike the security behavior of stored procedures, all

the procedures in the DBMS.SESSION package are run with privileges of the invoking user (and not privileges of the procedure or package owner). We can call these procedures first from a front end tool or Oracle Call Interface program, enable the proper roles via IS_ROLE_ENABLED and SET_ROLE, and then call URA97 procedures for assigning or revoking roles to a user. Of course, all of this will happen transparent to the end user.

In our implementation of URA97 a user invokes the stored procedure to grant or revoke a role from or to another user. The procedure calls are then as follows.

- ASSIGN(user, trole, arole)
- WEAK_REVOKE(user, trole, arole)
- STRONG_REVOKE(user, trole, arole)

The parameters user and trole (target role) specify which user is to be added to trole, or to be weakly or strongly revoked from trole. The arole parameter specifies which administrative role should be applied (with respect to the user who is invoking the URA97 procedure). We have included the arole parameter as a partial fix to the obstacle discussed above. The procedure code will of course check whether or not the user who calls the procedure is actually a member of arole.⁴

All the three procedures follow three basic steps.

1. If the user executing the procedure is an explicit or implicit member of arole then proceed to step 2, else stop execution and return an error message indicating this is not an authorized operation.
2. The tuple(s) from *can-assign* (for assign procedure) or *can-revoke* (for revocation procedures) are obtained where AR role value equals or is junior to the arole parameter specified in the procedure call.
3. If trole is in the specified range for any one of the tuples selected in step 2, then assign or revoke the trole else return an appropriate error message.

In case of ASSIGN also check whether the user being assigned to trole satisfies the prerequisite condition specified in the authorizing *can-assign* tuple or not.

In case of STRONG_REVOKE the operation may still fail due to all-or-nothing semantics.

⁴It is relatively straightforward to specify a set of administrative roles instead of a single arole, and we plan to extend our implementation to do that.

The implementation of steps 1 and 3 involves complex queries built on Oracle internal tables. These queries are performed dynamically at runtime. In order to check whether the user is a member of arole (in step 1) and whether the role is in the specified range for one of the relevant *can-assign* or *can-revoke* tuples (in step 3), we use Oracle CONNECT BY clause in our queries. By using CONNECT BY clause, one can traverse a tree structure corresponding to the role hierarchy in one direction. One can start from any point within the role hierarchy and traverse it towards junior or senior roles. But there is no control on the end point of the traversal. Specific branches or an individual node of the tree can be excluded by hard coding their values. Such hard coding is not appropriate for a general purpose stored procedure. In our implementation we overcome this problem by performing multiple queries and intersecting them to get the exact range. We specifically do not hard code any parameters in our queries.

In order to modularize our implementation we developed a package which performs the necessary checks involved in steps 1 and 3. All the procedures call this package to do the verification. The package contains several functions. Each one is designed to perform certain tasks, for example we have a function called *user_has_admin_role*. This function takes the parameters from the procedure which has called it and returns the results to the calling procedure. There are other functions which determine the range for a given arole.

Our implementation is convenient for the DBA since the stored procedures and packages we provide are generic and can be reused by other databases. The DBA only needs to define the roles and administrative roles, and configure the *can-assign* and *can-revoke* relations. Our implementation is available in the public domain for other researchers and practitioners to experiment with.

6 CONCLUSION

In this paper we have developed the URA97 model for assigning users to roles and revoking users from roles. URA97 is defined in context of the RBAC96 model [SCFY96]. However, it should apply to almost any RBAC model, including [FCK95, Gui95, GI96, HDT95, NO95], because user-role assignment is a basic administrative feature which will be required in any RBAC model.

Authorization to assign and revoke users to and from roles is controlled by administrative roles. The model requires users must previously satisfy a designated prerequisite condition (stated in terms of membership and non-membership in roles) before they can be enrolled via URA97 into additional roles. URA97 applies only to regular roles. Control of membership in administrative roles remains entirely in hands of the chief security officer. We have identified strong and weak revocation operations in URA97 and have defined their precise meaning.

The paper has also described an implementation of URA97 using Oracle stored procedures. Oracle's built in primitives are cumbersome to use for determining indirect membership in roles. We have implemented suitable functions and packages to enable this conveniently. These should be of use to other researchers and practitioners and are available in the public domain.

A significant hurdle we encountered is that Oracle does not allow a stored procedure to determine the roles that are turned on in a given session. This is a general problem of Oracle that will arise whenever we try to extend Oracle RBAC via stored procedures. In our implementation we require the user to specify these roles explicitly when the stored procedure is called. As discussed this could be made largely transparent with a suitable front end. Since most users will interact with Oracle via such a front end this may not be a significant problem in practice.

In future work we will extend URA97 to develop more comprehensive role-based administrative models encompassing administration of role-permission assignment and role-role relationships. We will also investigate how URA97 can be adapted for user-group assignment on platforms such as Unix and Windows NT (including simulation of group hierarchies which neither product provides). More generally we feel our work will inspire other researchers and developers to investigate administrative models in a systematic, scientific and experimental approach. We feel the security community has much to gain by pursuing such work.

Acknowledgment

This work is partially supported by the National Science Foundation and the National Security Agency.

References

- [BSJ93] Elisa Bertino, Pierangela Samarati, and Sushil Jajodia. Authorizations in relational database management systems. In *Proceedings of 1st ACM Conference on Computer and Communications Security*, pages 130–139, Fairfax, VA, November 3–5 1993.
- [Fag78] R. Fagin. On an authorization mechanism. *ACM Transactions on Database Systems*, 3(3):310–319, 1978.
- [FCK95] David Ferraiolo, Janet Cugini, and Richard Kuhn. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 241–48, New Orleans, LA, December 11–15 1995.
- [Feu95] Steven Feuerstein. *Oracle PL/SQL Programming*. O'Reilly & Associates, Inc., 1995.
- [FK92] David Ferraiolo and Richard Kuhn. Role-based access controls. In *Proceedings of 15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, October 13–16 1992.
- [FWF95] Eduardo B. Fernandez, Jie Wu, and Minjie H. Fernandez. User group structures in object-oriented database authorization. In J. Biskup, M. Morgernstern, and C. Landwehr, editors, *Database Security VIII: Status and Prospects*. North-Holland, 1995.
- [GI96] Luigi Guiri and Pietro Iglio. A formal model for role-based access control with constraints. In *Proceedings of IEEE Computer Security Foundations Workshop 9*, pages 136–145, Kenmare, Ireland, June 1996.
- [GSF91] Ehud Gudes, Haiyan Song, and Eduardo B. Fernandez. Evaluation of negative, predicate, and instance-based authorization in object-oriented databases. In S. Jajodia and C.E. Landwehr, editors, *Database Security IV: Status and Prospects*, pages 85–98. North-Holland, 1991.

- [Gui95] Luigi Guiri. A new model for role-based access control. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 249–55, New Orleans, LA, December 11–15 1995.
- [GW76] P.P. Griffiths and B.W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems*, 1(3):242–255, 1976.
- [HDT95] M.-Y. Hu, S.A. Demurjian, and T.C. Ting. User-role based security in the ADAM object-oriented design and analyses environment. In J. Biskup, M. Morgernstern, and C. Landwehr, editors, *Database Security VIII: Status and Prospects*. North-Holland, 1995.
- [KL95] George Koch and Kevin Loney. *Oracle The Complete Reference*. Oracle Press, 1995.
- [Lun88] Teresa Lunt. Access control policies: Some unanswered questions. In *Proceedings of IEEE Computer Security Foundations Workshop II*, pages 227–245, Franconia, NH, June 1988.
- [MD94] Imtiaz Mohammed and David M. Dilts. Design for dynamic user-role-based security. *Computers & Security*, 13(8):661–671, 1994.
- [NO95] Matunda Nyanchama and Sylvia Osborn. Access rights administration in role-based security systems. In J. Biskup, M. Morgernstern, and C. Landwehr, editors, *Database Security VIII: Status and Prospects*. North-Holland, 1995.
- [RBKW91] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1), 1991.
- [San97a] Ravi Sandhu. Rationale for the RBAC96 family of access control models. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*. ACM, 1997.
- [San97b] Ravi Sandhu. Roles versus groups. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*. ACM, 1997.
- [SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.
- [vSvdM94] S. H. von Solms and Isak van der Merwe. The management of computer security profiles using a role-oriented approach. *Computers & Security*, 13(8):673–680, 1994.
- [YCS97] Charles Youman, Ed Coyne, and Ravi Sandhu, editors. *Proceedings of the 1st ACM Workshop on Role-Based Access Control, Nov 31-Dec. 1, 1995*. ACM, 1997.

Short Talks

Advanced Internet Search Tools: Trick or Threat?

G. Lorenz S. Dangi D. Jones P. Carpenter G. Manes S. Shenoi *

Department of Computer Science
Keplinger Hall, University of Tulsa
Tulsa, Oklahoma 74104-3189, USA

Current Internet search tools, e.g., Yahoo! and AltaVista, are relatively simple. Their reliance on indexed files containing keyword-to-IP-address mappings limits them to handling low-level keyword queries. Future Internet search tools will be much more sophisticated. They will employ metadata repositories to support content-based querying and distributed, persistent agents performing a variety of functions, including data gathering, metadata extraction, data mining and information fusion. Users could create swarms of persistent search agents that would range the Internet in response to sophisticated queries, keeping them informed about updates and terminating only on explicit user directives. Clearly, such search engines will pose serious threats to security and privacy.

This paper describes the architecture of an advanced search engine being developed at the University of Tulsa to evaluate security and privacy threats. The server houses a metadata repository, a base agent and various search agents.

The metadata repository maintains schema information about information repositories, including structured, semi-structured and unstructured sources. It is continually refreshed by metadata daemons, persistent agents that search for new information sources, old sources that are no longer accessible and those whose schemas have been modified.

The base agent manages all queries submitted to the search engine. It analyzes queries and determines the appropriate number and type of search agents needed. The base agent can start new search agent threads or spawn persistent search agents. A new thread is created for each non-persistent query. A persistent query, e.g., one involving a changing or evolving information source, requires the creation of a persistent agent. All search agents report to their base agent which forwards information to the user.

Search agents perform various functions, including data gathering, querying, data mining and information fusion. These agents access the metadata repository for intensional (schema) information about information sources. Separate translation agents, one for each type of information source, are used to obtain actual data. Search agents transmit results to their base agent. In addition, they provide progress reports conveying the number of sources searched, the number remaining to be searched and the estimated completion time. Data access failures are reported and inaccessible sources are flagged for possible elimination from the metadata repository. A user maintains control of search agents through their base agent, enabling searches to be tuned, suspended or terminated.

Translation agents enable search agents to access heterogeneous information sources. Requests from search agents, expressed in the Knowledge Query Manipulation Language (KQML), are translated into the native languages of information sources. Similar information sources, e.g., databases using the Java/Open Database Connectivity (JDBC/ODBC) Interface, require a single translation agent. Translation agents can reside on the server or on machines hosting information sources. Information source administrators wishing to facilitate and/or control access by search agents may choose to provide their own local translation agents.

Most search engine users will use a web browser to access a GUI and query processor implemented by Java applets, similar to current search engines. However, a copy of the base agent may be downloaded and run locally on a remote machine. The ability to remotely execute base agents enhances local and global performance. Note that the implementation of the base agent as a Java application allows full search engine functionality unhindered by the Java applet security model.

* Address correspondence to sujeet@utulsa.edu.

Research supported by MPO Contracts MDA904-94-C-6117 and MDA904-96-1-0114 and OCAST Grant AR2-002.

An Environment for Developing Securely Interoperable Heterogeneous Distributed Objects

M. Berryman C. Rummel M. Papa J. Hale J. Threet S. Shenoi *
Department of Computer Science
Keplinger Hall, University of Tulsa
Tulsa, Oklahoma 74104-3189, USA

This paper describes the implementation of the Meta-Object Operating System Environment (MOOSE) for supporting the development, execution and verification of secure heterogeneous distributed systems. Security features in MOOSE are integrated at the meta-level within a new coordination language for heterogeneous software components that interact in a distributed virtual machine.

MOOSE's hierarchical operational and verification frameworks blend formal methods and object technology. The foundation of MOOSE is provided by the Robust Object Calculus (ROC), a process calculus for modeling and reasoning about distributed objects. The Meta-Object Model (MOM) defined with ROC is a primitive distributed object architecture for constructing sophisticated object models and programming languages. MOM implements a capabilities-based security model of access control for distributed objects. Capabilities, which are unforgeable tokens, are modeled in ROC by unique names that are not visible and cannot be reproduced.

We have used MOM to design an object-based coordination language, Mumbo, for orchestrating the secure interoperability of heterogeneous resources in open systems. Mumbo employs wrapper technology and abstract specifications to integrate native components, while translators provide mappings from high-level languages to ROC, permitting source-level integration. Mumbo uses the MOM security model to support Discretionary Access Control (DAC) for software components. It also provides new language constructs for constraining class and object protocols, giving developers more control over component communication patterns.

The ROC Virtual Machine (ROCVM) has been developed in Java to execute (reduce) ROC expressions, simulating and executing applications in heterogeneous distributed environments. ROCVM's primary security responsibility is to protect unique names that model capabilities in MOM's security architecture. Users cannot reproduce unique names or even view them without permission. ROCVM provides an interactive graphical interface for intuitive visualization and analysis of systems. The Java implementation permits operation on heterogeneous platforms. ROCVM is designed for distributed operation, providing multiple viewports for users into a given system. Users can interact with ROCVM by adding ROC expressions to executing systems on the fly. Verification tools are also integrated into ROCVM, making high assurance for heterogeneous distributed systems more practical.

* Address correspondence to sujeet@utulsa.edu.

Research supported by MPO Contracts MDA904-94-C-6117 and MDA904-96-1-0115 and OCAST Grant AR2-002.

Multilevel Decision Logic

A Formalism for Multilevel Rules Mining

Xiaoling Zuo

Department of Computer Science
Shanghai Jiaotong University
Shanghai, P. R. China

and

T. Y. Lin*

Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

In this paper, we will set up MLS decision logic. Based on this logic, then we can formalize and then discuss inference rules in information tables which include view and relation instances in relational databases. Roughly, a formula in the logic is meant to describe "object" in information tables. Objects may have the same description; thus formulas may describe subsets of objects. For single level version we refer readers to (Palwak, 1991).

The Syntax of a MLS DL-language

Alphabet

- a) T – The set of attribute names
- b) $V = \cup \text{dom}(A)$ – The set of attribute values of $A \in T$, called active domain of a . (Meyer, 1983)
- c) $\Xi = \{\neg, \wedge, \vee, \rightarrow, \equiv\}$ – The set of connectives (negation, and, or, implication, equivalence)
- d) C – A symbol to create a "place holder" to hold security classes.

Formulas Ω

The smallest set satisfying the following:

- a) Expressions of the form, attribute value pair $\langle A, v \rangle$, called atomic formulas, are formula of DL-language for any $A \in T$ and $v \in \text{dom}(A)$.
- b) If ϕ and η are formulas, so are $\neg\phi$, $(\phi \wedge \eta)$, $(\phi \vee \eta)$, $(\phi \rightarrow \eta)$
- c) To each formula ϕ in DL-language, we associate a variable to hold a security class, denoted by $C(\phi)$.

The Semantics of a MLS DL-language

MLS DL-Model

An information table $S = (U, T)$ (also known as information system, knowledge representation system) consists of

- (1) $U = \{u, v, \dots\}$ is a set of entities.
- (2) T is a set of attributes $\{A_1, A_2, \dots, A_n\}$.
- (3) $\text{Dom}(A_i)$ is the set of values of attribute A_i .
 $\text{Dom} = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \dots \cup \text{dom}(A_n)$.
(In databases (Meyer, 1983) dom is commonly referred as active domain)
- (4) $\rho : U \times T \rightarrow \text{Dom}$, called description function, is a map such that $\rho(u, A_i)$ is in $\text{dom}(A_i)$ for all u in U and A_i in T .

* This research is partially supported by Electric Power Research Institute, Palo Alto, California

** On leave from San Jose State University (tylin@cs.sjsu.edu)

An information table is called a relation if for each u the associated map

$$t = \rho(u, \bullet) : U \rightarrow \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n); \quad u \rightarrow (\rho(u, A_1), \rho(u, A_2), \dots, \rho(u, A_n))$$

is a faithful representation of U .

A *decision table* is an information table in which the attribute set $T = C \cup D$ is a union of two non-empty sets, C and D , of attributes. The elements in C are called conditional attributes. The elements in D are called decision attributes.

Interpretations

As usual at each level, we will denote $u \models_S \phi$ or $u \models \phi$, when S is understood, if an object $u \in U$ satisfies a formula ϕ in a information table $S=(U, T)$. So we will say $u \models \phi$, iff

$$\begin{aligned} u \models \langle A, v \rangle &\text{ iff } \rho(u, A) = v \\ u \models \neg \phi &\text{ iff non } u \models \phi \\ u \models (\phi \wedge \eta) &\text{ iff } u \models \phi \text{ and } u \models \eta \\ u \models (\phi \vee \eta) &\text{ iff } u \models \phi \text{ or } u \models \eta \end{aligned}$$

We have many usual formulas, such as

$$u \models (\phi \rightarrow \eta) \text{ iff } u \models \neg \phi \vee \eta$$

We associate the formula ϕ , the following set

$$I \phi I_S = \{ u : u \in U \text{ and } u \models_S \phi \}.$$

It will be called the *meaning* of ϕ . A formula is said to be *true* if $I \phi I_S = U$; ϕ is *logically equivalent* to η iff their meanings are the same, i.e., $I \phi I_S = I \eta I_S$. All formula and their meanings are properly classified.

Monotonic assumption. We will assume at each level (*the level and its dominated levels*) the Universe U is "well defined," often we may need to use U^L to denote the level L . We will assume $U^L \subseteq U^H$, where $L \leq H$

The Deductive System of a MLS DL-language

At each security level (more precisely, the level and its dominated levels), we have

The inference rules: Modus ponens is the only rule.

The axioms

- (1) The set of propositional tautologies
- (2) Specific axioms:

- (a) $\langle A, v \rangle \wedge \langle A, u \rangle \equiv 0$ for any $A \in T$ and $v, u \in V$ and $v \neq u$
- (b) $\bigvee \{ \langle A, v \rangle : \text{for every } v \in \text{dom}(A) \text{ and for every } A \in T \} \equiv 1$
- (c) $\neg \langle A, v \rangle \equiv \bigvee \{ \langle A, u \rangle : \text{for every } u \in \text{dom}(A) \text{ and for every } A \in T, v \neq u \}$

We need few auxiliary notations and results: Let 0 and 1 denote falsity and truth at every security level.. Formula of the form

* This research is partially supported by Electric Power Research Institute, Palo Alto, California

** On leave from San Jose State University (tylin@cs.sjsu.edu)

$$\langle A_1, v_1 \rangle \wedge \langle A_2, v_2 \rangle \wedge \dots \wedge \langle A_n, v_n \rangle$$

is called P-basic formula or P-formula, where $v_i \in \text{dom}(A_i)$, and $P = \{ A_1, A_2, \dots, A_n \}$. For $P = T$, P-basic formulas will be called basic formulas. The set of all basic formulas satisfiable in S is called *basic knowledge* in S . The specific Axiom a) follows from the assumption that each entity can have exact one value in each attribute. The Axiom b) must take one of the value of its domain. This is saying that $\text{dom}(A)$ is the active domain of attribute A . The axiom c) allow us to get rid of the negation in such a way that instead of saying that an object does not possesses a given property we can say that it has one of the remaining properties. It implies the closed word assumption. Let $\Sigma_S(P)$, or simply $\Sigma(P)$ denote the disjunction of all P-basic formulas satisfied in S . At each level, the closed word assumption can be express in the following (Pawlak, 1991).

Proposition $\models_S \Sigma_S(P) \equiv 1$. For any $P \subseteq T$.

A formula ϕ is a theorem, denoted by $\vdash \phi$, if it is derivable from the axioms.

At each level, the set of theorems of DL-logic is *identical with the set of theorems of classical propositional calculus with specific axioms (a)- (c)*. Computational rules of security classes of formulas with respect to logic connectives will be discussed in the full paper.

* This research is partially supported by Electric Power Research Institute, Palo Alto, California

** On leave from San Jose State University (tylin@cs.sjsu.edu)